

Trabajo Fin de Máster
Máster Universitario en Ingeniería Industrial

**Diseño e implementación de un
sistema de control para la representación
gráfica a partir de imágenes**

MEMORIA

Autor: Lois Rilo Antelo
Director: Manel Velasco García
Convocatoria: Septiembre de 2016



Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona



Resumen

En este trabajo Fin de Máster se pretende construir un robot que sea capaz de dibujar en una pizarra, a través de un rotulador situado en su actuador, realizando una representación de una imagen digital que se recibe como argumento de entrada.

Para ello se empleará un robot delta, que es un tipo de robot paralelo de gran aplicación en la industria y en la impresión 3D. Un robot delta consta de tres cadenas cinemáticas iguales que unen dos bases, una fija (que da soporte a la estructura completa del robot) y otra móvil (que hará de actuador).

Estos robots tienen una estructura polar en la disposición de sus brazos, con una distancia angular de 120° entre todos ellos, que le otorga una simetría muy particular y muy útil a la hora de realizar cálculos para hallar la posición de sus articulaciones. Se analizarán en este documento, por tanto, las cinemáticas directa e inversa, de este tipo de robots y se hará uso de ella durante el diseño del robot y de su control.

Será necesario, en consecuencia, diseñar mecánicamente un robot y llevar a cabo su fabricación y su montaje. Para crear el robot se optará por diseñar y fabricar gran parte de las piezas a través de la impresión 3D, mientras que los demás componentes seleccionados serán elementos reutilizados y tres motores paso a paso con sus tres controladoras correspondientes cuya elección se justificará convenientemente para acabar describiendo el montaje del robot, consiguiendo así un diseño económico, modular y sencillo.

También se pretende de forma paralela, diseñar un algoritmo que se encargue de generar la trayectoria del robot a partir de una imagen digital y diseñar el control del robot para convertir esta trayectoria en un movimiento real.

Este control se realizará a través de una placa Arduino que se comunica con un ordenador gracias a ROS (*Robot Operating System*). El ordenador se encargará de generar las trayectorias en varios pasos: tratamiento de las imágenes digitales de entrada para poder extraer información de ellas, extracción de las curvas y trazos presentes en la imagen, generación de la trayectoria que deberá seguir el actuador del robot y traducción de esta trayectoria a trenes de pulsos convenientemente sincronizados que se aplicarán a los motores paso a paso que se encargan de mover los tres brazos del robot delta.

Arduino se encargará de ejecutar el movimiento, ordenando a las controladoras a través de pulsos de control provocar el avance o retroceso en un paso de los motores que tienen conectados. El pequeño microcontrolador recibirá estos datos a través de mensajes de ROS y los ejecutará. Estos mismos mensajes serán utilizados para mover un modelo en URDF del robot y poder evaluar su movimiento durante el diseño del algoritmo de control y después para ofrecer una visualización simultánea del movimiento del sistema real.

Finalmente se mostrarán resultados reales, dibujos realizados por el robot de forma autónoma, y se discutirán sus virtudes y defectos, tratando de dar justificación a estos últimos para indicar posibles líneas futuras de mejora.

Palabras clave: Robot, robótica, delta, Arduino, ROS, cinemática, motores paso a paso, dibujo, imágenes digitales, control.

Índice

Resumen	1
1. Introducción	5
1.1. Motivación	5
1.2. Objetivos	5
1.3. Alcance	6
2. Estado del arte	7
2.1. Robótica	7
2.2. Robots Delta	9
3. Modelo cinemático robot delta	11
3.1. Cinemática inversa	11
3.2. Cinemática directa	14
3.3. Funciones diseñadas	16
4. Componentes del robot	23
4.1. Diseño mecánico	23
4.1.1. Diseño de piezas para impresión 3D	24
4.1.2. Otras piezas y componentes	28
4.2. Motores paso a paso	29
4.2.1. Motor paso a paso empleado	31
4.3. Controladores de los motores	33
4.3.1. Opciones posibles	33
4.3.2. Primera selección	33
4.3.3. Segunda selección	34
4.4. Montaje	38
5. Creación de trayectorias	41
5.1. Tratamiento de imágenes	41
5.2. Extracción de curvas de Bézier	44
5.3. Generar recorrido	47
5.4. Conversión a trenes de pulsos	52
6. Comunicación con ROS	55
6.1. Paquete de control del robot	55
6.2. Paquete roserial y Arduino	59
6.3. Paquete de visualización	65
7. Resultados	78
8. Planificación temporal y costes	81
8.1. Planificación temporal	81
8.2. Costes	81
9. Conclusiones	83
Agradecimientos	84

Bibliografía	85
Referencias	85
Bibliografía Complementaria	86

1. Introducción

En el presente proyecto se pretende construir un robot paralelo que sea capaz de trasladar la información de una fotografía digital a un dibujo sobre una pizarra con su actuador, un rotulador o similar. Se busca a través de este proyecto adquirir y desarrollar conocimientos y habilidades en distintos campos relacionados con la robótica.

Para desarrollar el trabajo se dispone del laboratorio de Sistemas Distribuidos de Control en la Facultad de Matemáticas y Estadística (FME), donde se contará con un escritorio de trabajo así como con material y herramientas básicas.

1.1. Motivación

Este proyecto se ve motivado por la inquietud en adquirir nuevas habilidades y desarrollar un trabajo de diseño completo. Se pretende abarcar un abanico amplio de temas relacionados con la robótica y el diseño robótico, pudiendo de este modo adquirir una mayor especialización en esos campos. Entre otros se adquirirán y desarrollarán las siguientes habilidades:

- **Desarrollo de algoritmos lógicos para el control robótico:** Será necesario realizar programación en distintos lenguajes y con estructuras de nivel medio, lo que provocará un desarrollo de las habilidades del alumno en unos lenguajes (por ejemplo, Matlab) y el descubrimiento de otros con los que nunca se había trabajado (como puede ser Python).
- **Diseño de comunicaciones en sistemas robóticos:** Para comunicar los distintos componentes que intervienen en el movimiento de un robot y el intercambio de datos entre estos. El alumno se verá obligado a iniciarse en ROS (*Robot Operating System*).
- **Diseño mecánico:** El robot será diseñado desde cero, lo que implica un trabajo de diseño de la estructura del mismo y de su fabricación y montaje. Se mejorarán las habilidades del alumno con programas de diseño 3D y se enfrentará a problemas de diversa índole durante la construcción del robot que se verá obligado a solucionar por sí mismo.
- **Trabajo con diversos archivos de texto plano:** Durante la realización de su trabajo el alumno se enfrentará a formas de organizar la información y de crear imágenes y diseños 3D nuevos para él como son por ejemplo las extensiones de archivos *.svg* y *.urdf*.

1.2. Objetivos

Los objetivos principales que se han planteado al inicio del proyecto son los siguientes:

- **Diseñar y construir un robot delta:** Se buscará un método sencillo y económico sin las exigencias que tendría la creación de un robot comercial.
- **Crear un algoritmo que controle el movimiento del robot:** Ser capaz de convertir la información de una imagen digital en datos que pueda interpretar el robot y sus actuadores transformándolos en el movimiento del robot a lo largo de una trayectoria.
- **Realizar dibujos a través del movimiento del robot:** El movimiento controlado del robot a través de una trayectoria debe de estar enfocado a producir dibujos con un rotulador sobre una pizarra (o un sistema de dibujo similar).

A la vez, también se han especificado una serie de objetivos secundarios, en parte consecuencia de los objetivos primarios, para enfocar más el trabajo en algunos aspectos:

- **Adquisición de conocimientos previos y preparación:** Conseguir un nivel básico suficiente para poder trabajar con distintos programas y lenguajes de programación que intervendrán en el trabajo, como son: ROS, FreeCAD, Python, C++...
- **Analizar y entender la cinemática de un robot delta:** Tanto directa como inversa.
- **Idear un método para extraer información de una imagen digital:** Poder extraer información de una imagen es el primer paso para poder dibujar lo que aparece en la misma.
- **Diseñar un método de comunicación:** Las distintas partes del robot deben comunicarse para poder funcionar. Será necesario entender ROS y el funcionamiento de sus nodos, temas y mensajes para poder realizar una comunicación efectiva.

1.3. Alcance

El alcance de este proyecto por tanto incluye contruir y diseñar un robot, diseñar un algoritmo para su movimiento, montar el robot y realizar dibujos con él. Sin embargo cabe aclarar algunos puntos que quedan fuera del alcance del mismo:

- El desarrollo de un algoritmo eficiente para la extracción de los contornos más adecuados de una imagen para realizar luego su dibujo. El desarrollo de este tipo de algoritmos es un trabajo muy amplio y que podría constituir un poryecto en si mismo, por esta razón se opta por dejarlo fuera del alcance del presente trabajo.
- Conseguir un robot con un movimiento extremadamente preciso y que consiga reproducir los dibujos con gran exactitud. Conseguir un robot que dibuje una imagen tal cual se ve requiere grandes esfuerzos de diseño y resideño, así como gasto en mejores componentes y materiales, es por esta razón que se considerará que una gran exigencia en cuanto a resultados exigiría ampliar los márgenes temporales del proyecto y queda por tanto fuera del alcance.

2. Estado del arte

En esta sección se realizará un breve repaso al estado de las tecnologías y ciencias relacionadas con este proyecto.

2.1. Robótica

Un robot es un dispositivo (o grupo de dispositivos) electromecánico o biomecánico que pueden realizar tareas de forma autónoma (bajo el control de un ordenador preprogramado) o semiautónoma (bajo el control directo de un ser humano)[1] [2].

La robótica es la ciencia y la tecnología que se ocupa del estudio y funcionamiento de los robots a través del diseño, manufactura y aplicación de estos. A su vez, la robótica combina diversas disciplinas como la mecánica, la electrónica, la informática, la inteligencia artificial y la ingeniería de control.

Existen muchos tipos de robots, desde robots manipuladores y robots móviles hasta robots con mecanismos más complejos y robots inspirados en animales o humanos.



Figura 1: Robot móvil (a) y robot humanoide [3](b)

Ante esta gran variedad de robots es lógico pensar que también existen innumerables aplicaciones para ellos, y de hecho así es. Uno de los campos donde podemos encontrar muchos robots es en la industria, en muchos casos estos serán robots manipuladores que se encargan de tareas como la soldadura, el ensamblaje, el pintado, la transferencia de material... e incluso algunos que trabajan en el mismo espacio que trabajadores humanos, asistiendo a estos en sus tareas [4].

También podemos encontrar robots que realizan trabajos de campo muy diversos y en muy distintas áreas, enumeramos algunos ejemplos como muestra de los amplios que son las aplicaciones de campo de la robótica.

- Robots submarinos: instalaciones de gaseoductos a grandes profundidades, recogida de muestras marinas, investigaciones militares, arqueológicas...
- Robots aéreos: Vigilancia y recogida de datos, asistencia sanitaria rápida, transporte (paquetes e incluso pasajeros)...
- Robots espaciales: principalmente se emplean para la exploración de la superficie de planetas y satélites, y como robots orbitantes que realizan operaciones a distancia.
- Robots agrícolas y forestales: Recogida de las cosechas, explotación forestal y recogida de información del suelo son las tareas mayoritarias en estos campos.

- Robots constructores: Levantamiento de muros, colocación de elementos prefabricados, pintado, corte de materiales, tareas de acabado e incluso de rotura y demolición.
- Robots de rescate: Asisten en tragedias medioambientales como terremotos o tsunamis y ayudan en el rescate de personas en zonas de difícil acceso y/o de alto riesgo.



(a)



(b)

Figura 2: Colaboración humano y robot (a) y robot espacial destinado a Marte [5] (b)

En la actualidad también podemos encontrar cada vez más presencia de la robótica en entornos médicos. Muchos robots asisten en cirugías y operaciones o ayudan en tareas de rehabilitación, de hecho últimamente se está avanzando mucho en cuanto a prótesis robóticas que sustituyen alguna extremidad [6] y exoesqueletos que asisten al movimiento en personas con dificultades motoras o en tareas que requieren bastante fuerza física[7].

Vivimos en un mundo en el que la tecnología evoluciona a pasos de gigante, y es que cada vez será más habitual encontrar robots domésticos e incluso realizando interacciones sociales, como Tibi y Dabo, dos robots desarrollados por el IRI¹. Los robots domésticos pueden realizar tareas de limpieza y mantenimiento de forma autónoma (fregar el suelo, limpiar la piscina, cortar el césped, limpiar las ventanas). Y es que esto es solo el comienzo, los robots aparecerán también en entornos sociales realizando tareas de recepción en hoteles, guiado de turistas o simplemente de entretenimiento (robots que imitan el comportamiento de una mascota).



(a)



(b)

Figura 3: Prótesis robótica (a) y Tibi y Dabo, robots urbanos desarrollado por el IRI (b)

¹El Institut de Robòtica i Informàtica Industrial es un centro de investigación de la UPC

2.2. Robots Delta

Un robot delta es un tipo concreto de robot manipulador paralelo. Un robot paralelo está compuesto por varias cadenas cinemáticas cerradas en un actuador final, o plataforma móvil. La base fija y la base móvil se unen a través de estas cadenas cinemáticas, que de forma independiente son cadenas en serie, y en general son simétricas para simplificar los cálculos. Típicamente cada brazo está controlado por un actuador y en general estos robots paralelos pueden manipular una carga mayor que los robots de cadena abierta al compartir la carga entre varios brazos.

Suelen ser robots ligeros, rígidos y capaces de imprimir grandes aceleraciones al actuador. Sus aplicaciones van desde simulaciones de vuelo hasta a la transferencia de piezas a alta velocidad en la industria.

Es probable que el primer desarrollo técnico de este tipo de robots corriese a cargo de James Gwinnett en 1931 [8], pero no sería hasta una década después cuando Williard Pollard intentó un nuevo robot paralelo con una aplicación en la industria, automatizar el pintado con spray. Ingenieros como E. Gough y D. Stewart continuaron evolucionando con este tipo de robot, llegando a idear lo que ahora se conoce como plataforma Stewart.

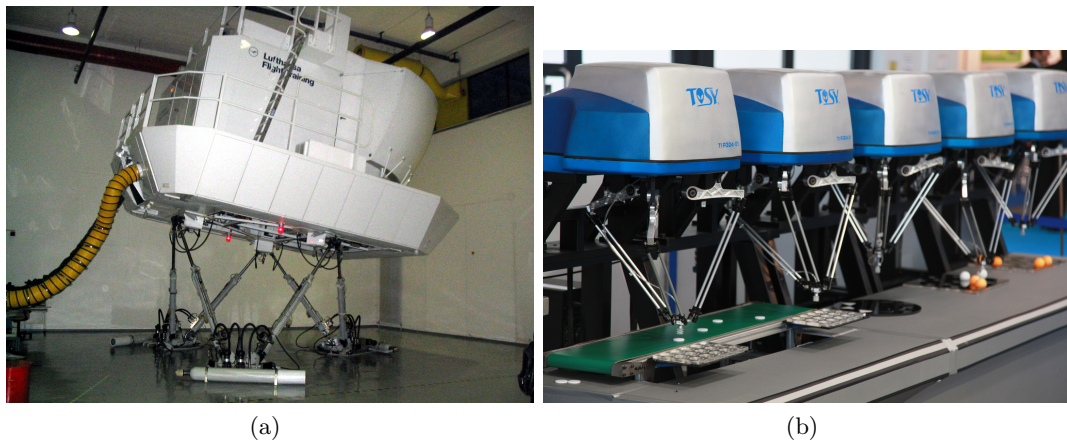


Figura 4: Robot paralelo usado en un simulador de vuelo (a) y robot paralelo tipo delta industrial(b)

El robot delta fue inventado en la década de 1980 por un grupo de investigación liderado por el profesor Reymond Clavel en l'École Polytechnique Fédérale de Lausanne (EPFL, Suiza). Debido a las necesidades de la industria, el propósito de este nuevo tipo de robot paralelo fue manipular objetos ligeros y pequeños a grandes velocidades. El robot paralelo tipo delta, mostrado en la Figura 5a, es simétrico, espacial y compuesto por tres eslabonamientos idénticos los cuales conectan la base fija con la plataforma móvil [9]. Cuenta con 3 o 4 grados de libertad, dependiendo de si permite la rotación de su base móvil o no.

En 1987, Demarex (una compañía suiza) adquirió la licencia del robot delta y comenzó la producción de robots delta destinados a la tareas de empaquetamiento. Años más tarde, ya en 1991, Clavel presentó su tesis doctoral, titulada *Conception d'un robot parallèle rapide à 4 degrés de liberté* [10]. Años después (en 1999) recibió el premio del robot dorado reconociendo sus esfuerzos en el desarrollo del robot delta. Mientras que ese mismo año ABB comenzó también a comercializar su propio robot delta, el FlexPicker, al igual que Sigpack Systems. En la actualidad se siguen sacando nuevos robot delta y nuevas marcas entran al mercado, como FANUC que ha diseñado una serie de robots delta con capacidad para distintas cargas.

En un robot delta la base y el actuador final se conectan por tres cadenas cinemáticas cerradas e idénticas. Cada cadena esta separada 120° una de otra, y tiene un brazo superior y un sistema de barras paralelas. Esta configuración restringe los movimientos del actuador

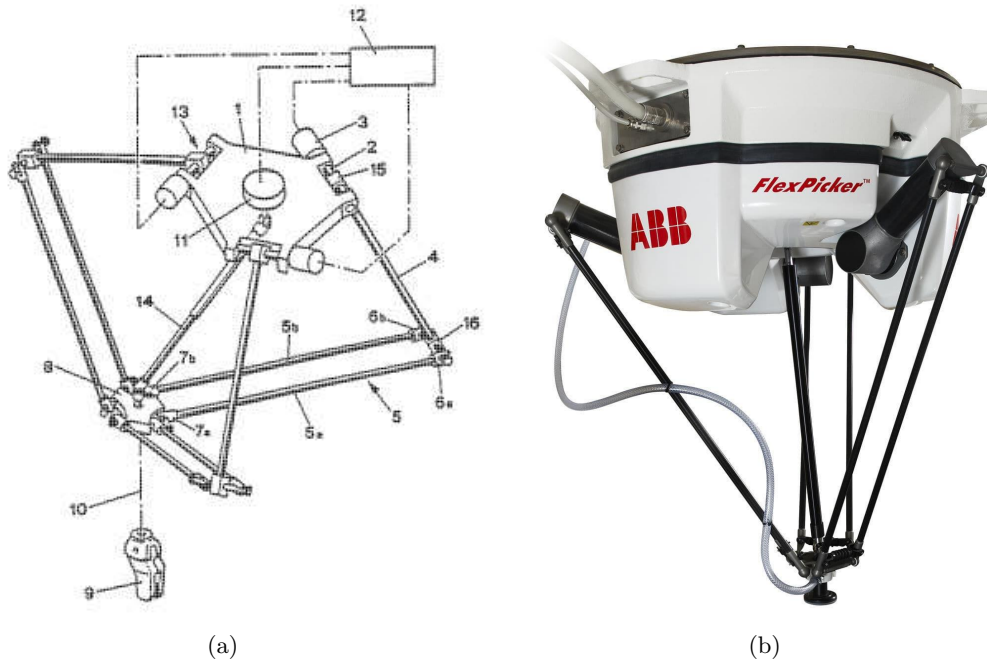


Figura 5: Robot delta diseñado por Clavel(a) y FlexPicker de ABB(b)

a 3 traslaciones de acuerdo con los ejes X, Y, Z. En cada cadena cinemática, la base y el elemento final son conectados con una articulación rotacional y 4 articulaciones esféricas [11].

Los motores están montados sobre la base, y transfieren el movimiento a cada articulación rotacional. Este montaje permite que la carga inercial manejada sea reducida. Un cuarto grado de libertad puede ser añadido mediante un brazo central formado por 2 articulaciones universales, una prismática y una rotacional. Este grado permitirá girar la paleta para la aplicación deseada.

3. Modelo cinemático robot delta

Al querer construir nuestro propio robot delta se nos presentan dos problemas cinemáticos: conocer la posición del actuador final en coordenadas cartesianas XYZ a partir de la posición angular de cada uno de los tres brazos del robot (cinemática directa) y conocer en que posición deberíamos colocar los tres brazos del robot para colocar nuestro actuador final en la posición deseada (cinemática inversa).

A continuación se presenta un esquema cinemático del robot delta, Figura 6. Las plataformas superior e inferior son dos triángulos equiláteros. La superior (y visualmente más grande) está fija y en ella se sitúan los motores que mueven las articulaciones de cada brazo, dando lugar a los ángulos θ_1, θ_2 y θ_3 , tal y como se observa en la figura. El triángulo inferior (de menor dimensión) es móvil y en su centro se encuentra el actuador final (E_0) descrito por la posición cartesiana (x_0, y_0, z_0) .

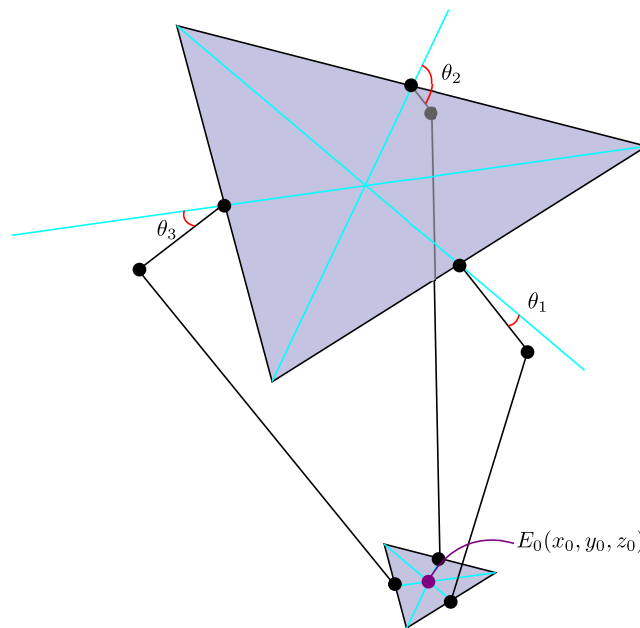


Figura 6: Esquema cinemático de un robot delta

3.1. Cinemática inversa

En primer lugar, vamos a presentar algunos parámetros importantes de la geometría de nuestro robot. Parámetros que serán determinados por el diseño final del robot:

- f designará al lado del triángulo fijo.
- e será el lado del triángulo móvil.
- r_f denotará la longitud del brazo superior.
- r_e corresponderá a la longitud del brazo inferior.

Además los ejes de coordenadas tendrán su origen en el centro de simetría del triángulo fijo, como se muestra en la figura 7, de este modo la coordenada Z del actuador será siempre positiva.

Por otro lado las uniones al triángulo fijo se denominarán F_i , las uniones entre los dos brazos (o codos) J_i y las uniones al triángulo móvil E_i . Siendo $i = 1, 2, 3$ ya que tenemos tres

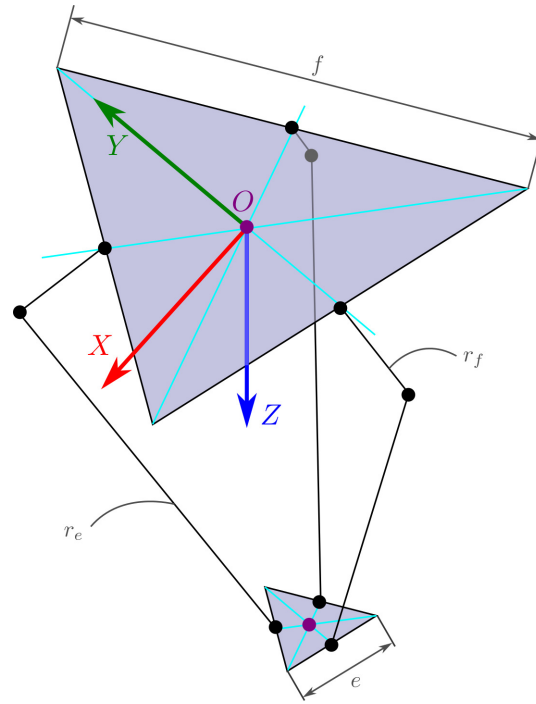


Figura 7: Parámetros de diseño

brazos exactamente iguales. Analizaremos ahora la cadena cinemática del brazo 1, teniendo en cuenta que con una simple rotación de los ejes de referencia llegaremos a las mismas ecuaciones para los otros brazos.

Debido al diseño del robot, el brazo superior F_1J_1 (ver Figura 8) solo puede rotar en el plano YZ , formando un círculo con centro en F_1 y radio r_f . De forma opuesta J_1 y E_1 son uniones universales, de modo que E_1J_1 puede rotar libremente con centro en E_1 , formando una esfera con centro en dicho punto y radio r_e .

De este modo, la intersección de esta esfera y el plano YZ será un círculo con centro en el punto E'_1 y radio E'_1J_1 , donde E'_1 es la proyección de E_1 en el plano YZ . El punto J_1 se puede hallar como la intersección de dos círculos conocidos con centros en E'_1 y F_1 . Hay dos soluciones, nos debemos de quedar la que tenga menor coordenada Y , ya que es la que tiene sentido en nuestra robot. Y dado que sabemos la posición de J_1 ya podemos calcular el ángulo θ_1 .

Ahora ya podemos, partiendo de la posición del actuador final $E_0(x_0, y_0, z_0)$, y realizando los siguientes cálculos trigonométricos basándonos en los explicado y en la figura 9, llegar a obtener θ_1 .

$$E_0E_1 = \frac{e}{2} \tan 30^\circ = \frac{e}{2\sqrt{3}} \quad (1)$$

$$E_1(x_0, y_0 - \frac{e}{2\sqrt{3}}, z_0) \Rightarrow E'_1(0, y_0 - \frac{e}{2\sqrt{3}}, z_0) \quad (2)$$

$$E_1E'_1 = x_0 \Rightarrow E'_1J_1 = \sqrt{E_1J_1^2 - E_1E'_1^2} = \sqrt{r_e^2 - x_0^2} \quad (3)$$

$$F_1(0, -\frac{f}{2\sqrt{3}}, 0) \quad (4)$$

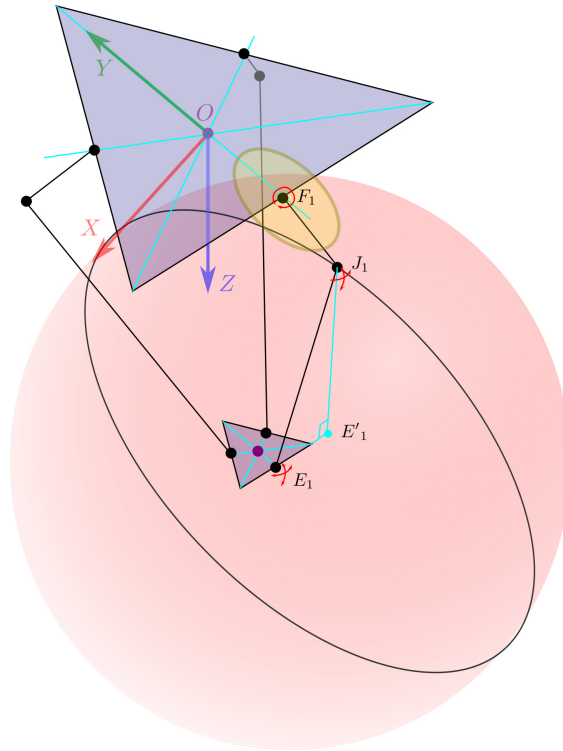


Figura 8: Uniones y cinemática de un brazo

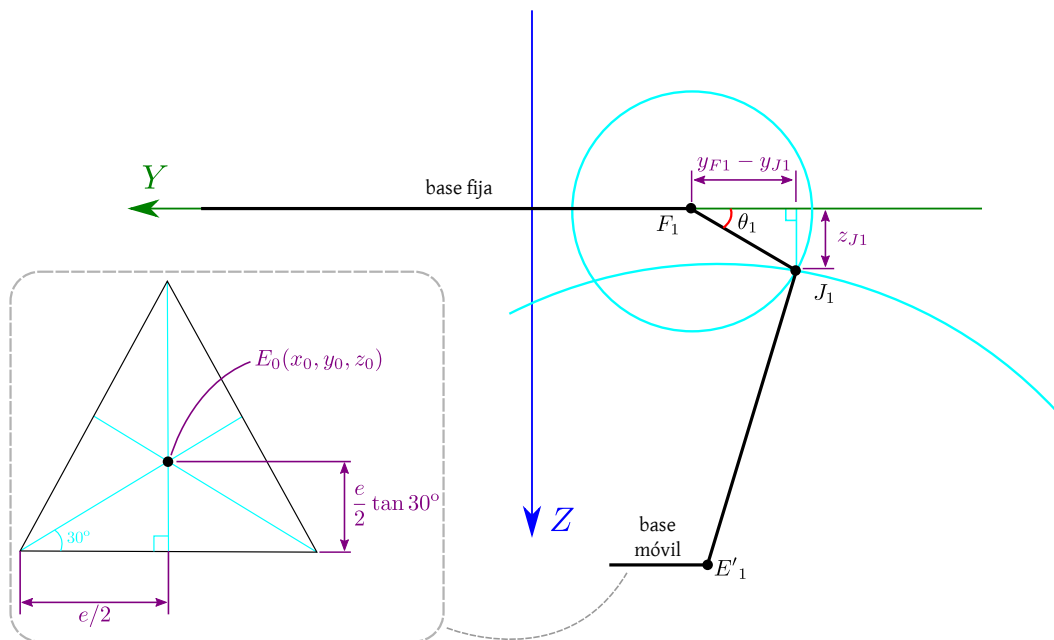


Figura 9: Puntos de interés para cálculos trigonométricos

Intersección de los dos círculos:

$$\begin{cases} (y_{J1} - y_{F1})^2 + (z_{J1} - z_{F1})^2 = r_f^2 \\ (y_{J1} - y_{E'1})^2 + (z_{J1} - z_{E'1})^2 = r_e^2 - x_0^2 \end{cases} \Rightarrow$$

$$\begin{cases} \left(y_{J1} + \frac{f}{2\sqrt{3}}\right) + z_{J1}^2 = r_f^2 \\ \left(y_{J1} - y_0 + \frac{e}{2\sqrt{3}}\right)^2 + (z_{J1} - z_0)^2 = r_e^2 - x_0^2 \end{cases} \Rightarrow J_1(0, y_{J1}, z_{J1}) \quad (5)$$

Y finalmente:

$$\theta_1 = \arctan\left(\frac{z_{J1}}{y_{F1} - y_{J1}}\right) \quad (6)$$

Esta simplicidad en los cálculos es debida a la colocación de los ejes de coordenadas, para seguir explotando esto en los ángulos restantes debemos hacer uso de la simetría del robot. Es decir, primero rotar el sistema de coordenadas en torno al eje Z 120° en el sentido contrario a las agujas del reloj para calcular con los nuevos ejes de referencia $X'Y'Z'$ el ángulo θ_2 con el mismo algoritmo usado antes para calcular θ_1 , y en segundo lugar rotar el sistema de coordenadas 120° en el sentido de las agujas del reloj para, de modo paralelo, calcular θ_3 .

3.2. Cinemática directa

En este caso los tres ángulos θ_1, θ_2 y θ_3 son dados y tenemos que determinar las coordenadas (x_0, y_0, z_0) del actuador final E_0 . Dado que sabemos los ángulos θ podemos obtener fácilmente las coordenadas de los puntos J_1, J_2 y J_3 . Los brazos inferiores pueden rotar libremente alrededor de los puntos J_1, J_2 y J_3 formando tres esferas de radio r_e .

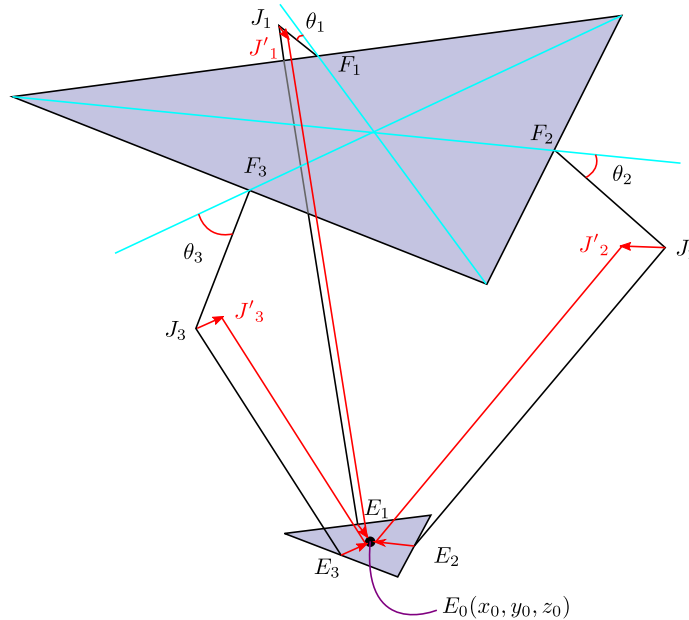


Figura 10: Cálculo cinemática directa

A continuación hacemos lo siguiente: movemos los centros de las esferas de los puntos J_1, J_2 y J_3 a los puntos J'_1, J'_2 y J'_3 con los vectores de traslación E_1E_0, E_2E_0 y E_3E_0 respectivamente. Después de esta traslación las tres esferas intersecarán en un punto: E_0 como se muestra en la figura 11.

Formalizando la idea en forma de ecuaciones encontramos que los puntos J'_1, J'_2 y J'_3 se pueden calcular de la siguiente manera:

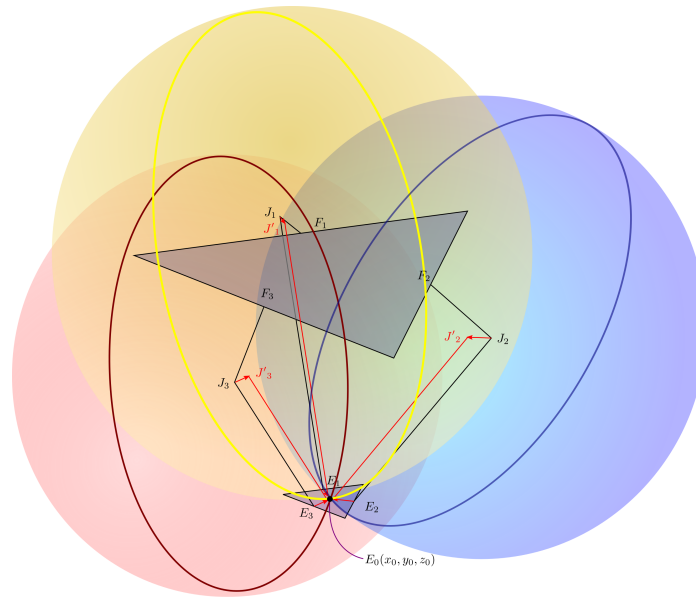


Figura 11: La intersección de las tres esferas es el punto del actuador

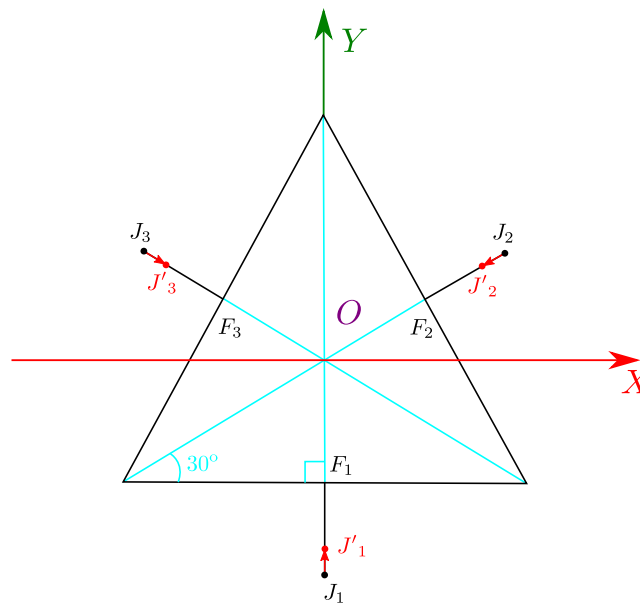


Figura 12: Proyección base superior sobre el plano XY y puntos de interés

$$OF_1 = OF_2 = OF_3 = \frac{f}{2} \tan 30 = \frac{f}{2\sqrt{3}} \quad (7)$$

$$J_1 J'_1 = J_2 J'_2 = J_3 J'_3 = \frac{e}{2} \tan 30 = \frac{e}{2\sqrt{3}} \quad (8)$$

$$F_i J_i = r_f \cos(\theta_i), \quad \forall i = 1, 2, 3 \quad (9)$$

De modo que nos quedan definidos de la siguiente forma:

$$J'_1 \left(0, -\frac{f-e}{2\sqrt{3}} - r_f \cos(\theta_1), -r_f \sin(\theta_1) \right) \quad (10)$$

$$J'_2 \left(\left[\frac{f-e}{2\sqrt{3}} + r_f \cos(\theta_2) \right] \cos 30, \left[\frac{f-e}{2\sqrt{3}} + r_f \cos(\theta_2) \right] \sin 30, -r_f \sin(\theta_2) \right) \quad (11)$$

$$J'_3 \left(\left[\frac{f-e}{2\sqrt{3}} + r_f \cos(\theta_3) \right] \cos 30, \left[\frac{f-e}{2\sqrt{3}} + r_f \cos(\theta_3) \right] \sin 30, -r_f \sin(\theta_3) \right) \quad (12)$$

Si las coordenadas de los puntos J_1 , J_2 y J_3 se designan como (x_1, y_1, z_1) , (x_2, y_2, z_2) y (x_3, y_3, z_3) respectivamente podemos escribir el siguiente sistema de ecuaciones con las ecuaciones de las tres esferas (nótese que $x_1 = 0$):

$$\begin{cases} x^2 + (y - y_1)^2 + (z - z_1)^2 = r_e^2 \\ (x - x_2)^2 + (y - y_2)^2 + (z - z_2)^2 = r_e^2 \\ (x - x_3)^2 + (y - y_3)^2 + (z - z_3)^2 = r_e^2 \end{cases} \quad (13)$$

Resolviéndolo obtenemos dos soluciones, de las cuales solo una será factible físicamente (la que tenga una coordenada z mayor) que será nuestra solución: $E_0(x, y, z)$.

3.3. Funciones diseñadas

Teniendo en cuenta lo explicado se han escrito varias funciones en Python y en Matlab para resolver la cinemática inversa y la cinemática directa del robot. Las funciones de Matlab se emplean en la sección 4 para tomar decisiones de diseño y valorar el espacio de trabajo del robot. Mientras que las funciones de Python se emplearán a lo largo de las secciones 5 y 6 para generar la trayectoria y mover el robot y la visualización del mismo.

Los siguientes listados de código muestran las funciones imprescindibles para calcular la cinemática inversa y directa de un robot delta en función de sus parámetros de diseño, se anima al lector a revisar el código completo que se encuentra en los archivos adjuntos a este trabajo.

Listado de código 1 Funciones para calcular la cinemática inversa y directa en Matlab

```
%FUNCION PARA CALCULAR LA CINEMATICA INVERSA:

function [theta1,theta2,theta3] = Inverse_Kinematics(EF)

%Parametros de disenho:
f = 125; %Lado triangulo superior (FF)
e = 130; %Lado triangulo inferior (EE)
global la
la = 150; %Brazo superior
global lb
lb = 205; %Longitud brazo inferior

%Parametros auxiliares
```

```

0 = [0,0,0];
hf = sqrt(0.75*(f^2)); %Altura FF
he = sqrt(0.75*(e^2));
%Matriz de rotacion en torno a eje Z:
ang = 120; ang=ang*pi/180;
Rz = [cos(ang) -sin(ang) 0;
      sin(ang)  cos(ang) 0;
      0         0       1];
%Puntos triangulo superior
A1 = [hf/3,0,0];
A2 = A1*Rz;
A3 = A2*Rz;

%Theta1:
EE1 = [EE(1)+he/3, EE(2), EE(3)]; %Punto de union en el lateral del
    triangulo movil.
%Circunferencia centro en X1p y radio lbp:
X1p = EE1; X1p(2)=0; %Proyeccion sobre plano XZ.
lbp = sqrt(lb^2-EE(2)^2);%Proyeccion radio.
%+Circunferencia centro en A1 y radio la:
%interseccion circunferencias:
x0 = [1; 1];
options = optimoptions(@lsqnonlin,'Display','off');
x = lsqnonlin(@(x) circunferencias(x,lbp,X1p,A1),x0,[A1(1), 0],[],options)
    ;
%sol:
B1 = [x(1), 0, x(2)];
theta1 = atan(B1(3)/(B1(1)-A1(1)))*180/pi; %grados

%theta2:
%Todos los puntos que ya existiesen se rotan 240 grados (sentido positivo)
EE120 = EE*Rz;
EE240 = EE120*Rz;
EE2 = [EE240(1)+he/3, EE240(2), EE240(3)];
%Circunferencia centro X2p y radio lbp
X2p = EE2; X2p(2) = 0;
lbp = sqrt(lb^2-EE240(2)^2);
%+Circunferencia centro en A2 y radio la:
%interseccion:
x0 = [1; 1];
A240=A2*Rz*Rz;
x = lsqnonlin(@(x) circunferencias(x,lbp,X2p,A240),x0,[A240(1), 0],[],
    options);
%sol:
B2 = [x(1), 0, x(2)];
theta2 = atan(B2(3)/(B2(1)-A240(1)))*180/pi; %grados

%theta3:
%Todos los puntos que ya existiesen se rotan 120 grados!
EE3 = [EE120(1)+he/3, EE120(2), EE120(3)];
%Circunferencia centro en X3p y radio lbp
X3p = EE3; X3p(2) = 0; %Proyeccion sobre plano XZ.
lbp = sqrt(lb^2-EE120(2)^2);
%+Circunferencia centro en A3 y radio la:
%interseccion:
x0 = [1; 1];
A120=A3*Rz;
x = lsqnonlin(@(x) circunferencias(x,lbp,X3p,A120),x0,[A120(1), 0],[],
    options);
%sol:
B3 = [x(1), 0, x(2)];
theta3 = atan(B3(3)/(B3(1)-A120(1)))*180/pi; %grados

```

```
end

%FUNCIONES AUXILIARES:

function F = circunferencias(x,lbp,Xp,A)

    global la
    %Circunferencia centro en Xp y radio lbp
    %Circunferencia centro en A y radio la
    %Plano XZ
    F=[(x(1)-Xp(1))^2 + (x(2)-Xp(2))^2 - lbp^2;
        (x(1)-A(1))^2 + (x(2))^2 - la^2];

end

function F = esferas(x)

    global lb
    global Pc
    F=[(x(1)-Pc(1,1))^2 + (x(2)-Pc(1,2))^2 + (x(3)-Pc(1,3))^2 - lb^2;
        (x(1)-Pc(2,1))^2 + (x(2)-Pc(2,2))^2 + (x(3)-Pc(2,3))^2 - lb^2;
        (x(1)-Pc(3,1))^2 + (x(2)-Pc(3,2))^2 + (x(3)-Pc(3,3))^2 - lb^2];
    if x(3)<0
        %solo nos valen soluciones de z positivo
        z=10000;
    end
    % F=[(x(1)-15)^2 + (x(2)-0)^2 + (x(3)-1.73)^2 - lb^2;
    %      (x(1)-(-6))^2 + (x(2)-(-10))^2 + (x(3)-7.66)^2 - lb^2;
    %      (x(1)-(-7))^2 + (x(2)-(13))^2 + (x(3)-3.42)^2 - lb^2];
end

%FUNCION PARA CALCULAR LA CINEMATICA DIRECTA:

function EE = End_Efector(theta1,theta2,theta3)

%Kinematic plot delta robot:

%Parametros de disenho:
f = 125; %Lado triangulo superior (FF)
e = 130; %Lado triangulo inferior (EE)
la = 150; %Brazo superior,200
global lb
lb = 205; %Longitud brazo inferior,350

O = [0,0,0];
hf = sqrt(0.75*(f^2)); %Altura FF
he = sqrt(0.75*(e^2));

%Matriz de rotacion en torno a eje Z:
ang = 120; ang=ang*pi/180;
Rz = [cos(ang) -sin(ang) 0;
      sin(ang)  cos(ang) 0;
      0         0      1];

O1 = [-2*hf/3,0,0];
O2 = O1*Rz;
O3 = O2*Rz;
```

```
A1 = [hf/3,0,0];
A2 = A1*Rz;
A3 = A2*Rz;

%Rotacion Brazos superiores:
%theta1 = 10;
theta1 = theta1*pi/180;
%theta2 = 10;
theta2 = theta2*pi/180;
%theta3 = 10;
theta3 = theta3*pi/180;

B1 = [A1(1)+la*cos(theta1) ,0 ,la*sin(theta1)];
B2 = [A1(1)+la*cos(theta2) ,0 ,la*sin(theta2)];
B2 = B2*Rz;
B3 = [A1(1)+la*cos(theta3) ,0 ,la*sin(theta3)];
B3 = B3*Rz; B3 = B3*Rz;
%desplazamos B a Bp con la distancia de los lados del triangulo movil a su
%centro para poder calcular la interseccion con esferas.
vhe = [-he/3 0 0]; %vector de desplazamiento al centro del triangulo del
EE.
B1p = B1 + vhe;
vhe = vhe*Rz;
B2p = B2 + vhe;
vhe = vhe*Rz;
B3p = B3 + vhe;
%Asi la matriz Pc, que define la posicion de los tres codos desplazados
queda:
global Pc
Pc = [B1p; B2p; B3p];

%Ahora desde los codos las varillas se mueven libres, por lo que se define
%una esfera en cada codod de radio lb, la interseccion de las tres esferas
%dara las 2 posiciones posibles, de las cuales una sera descartada por ser
%fisicamente imposible de alcanzar.
%ec. esfera: (x-xo)^2 + (y-yo)^2 + (z-zo)^2 = r^2

%Solucion:
x0 = [0; 0; 500];
options = optimoptions(@lsqnonlin,'Display','off');
x = lsqnonlin(@esferas,x0,[-500,-500,0],[],options);

EE=x';

end
```

Gracias a estas funciones podemos obtener toda la información sobre la posición de las distintas articulaciones y puntos del robot delta por lo que resulta sencillo realizar una representación simple del mismo en Matlab, figura 13. Las funciones para realizar esta representación están disponibles en los archivos adjuntos.

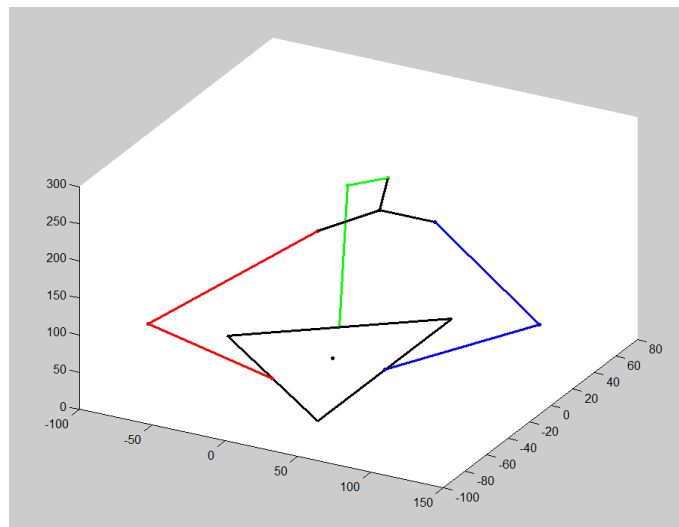


Figura 13: Representación de la posición del robot a través de Matlab

Listado de código 2 Funciones para calcular la cinemática directa e inversa a través de Python

```
# Original code from
# http://forums.trossenrobotics.com/tutorials/introduction-129/delta-robot
#   -kinematics-3276/
# License: MIT

import math

# Specific geometry for my delta robot:
e = 130.0 # small triangle side (EE)
f = 125.0 # support triangle side
re = 205.0 # lower arm length
rf = 150.0 # upper arm length

# Trigonometric constants
s = 165 * 2
sqrt3 = math.sqrt(3.0)
pi = 3.141592653
sin120 = sqrt3 / 2.0
cos120 = -0.5
tan60 = sqrt3
sin30 = 0.5
tan30 = 1.0 / sqrt3

# Forward kinematics: (theta1, theta2, theta3) -> (x0, y0, z0)
#   Returned {error code, theta1, theta2, theta3}
def forward(theta1, theta2, theta3):
    x0 = 0.0
    y0 = 0.0
    z0 = 0.0

    t = (f - e) * tan30 / 2.0
    dtr = pi / 180.0 # degrees to radians

    theta1 *= dtr
    theta2 *= dtr
    theta3 *= dtr
```

```

y1 = -(t + rf * math.cos(theta1))
z1 = -rf * math.sin(theta1)

y2 = (t + rf * math.cos(theta2)) * sin30
x2 = y2 * tan60
z2 = -rf * math.sin(theta2)

y3 = (t + rf * math.cos(theta3)) * sin30
x3 = -y3 * tan60
z3 = -rf * math.sin(theta3)

dnm = (y2 - y1) * x3 - (y3 - y1) * x2

w1 = y1 * y1 + z1 * z1
w2 = x2 * x2 + y2 * y2 + z2 * z2
w3 = x3 * x3 + y3 * y3 + z3 * z3

# x = (a1*z + b1)/dnm
a1 = (z2 - z1) * (y3 - y1) - (z3 - z1) * (y2 - y1)
b1 = -((w2 - w1) * (y3 - y1) - (w3 - w1) * (y2 - y1)) / 2.0

# y = (a2*z + b2)/dnm
a2 = -(z2 - z1) * x3 + (z3 - z1) * x2
b2 = ((w2 - w1) * x3 - (w3 - w1) * x2) / 2.0

# a*z^2 + b*z + c = 0
a = a1 * a1 + a2 * a2 + dnm * dnm
b = 2.0 * (a1 * b1 + a2 * (b2 - y1 * dnm) - z1 * dnm * dnm)
c = (b2 - y1 * dnm) * (b2 - y1 * dnm) + b1 * b1 + dnm * dnm * (z1 * z1 - re * re)

# discriminant
d = b * b - 4.0 * a * c
if d < 0.0:
    return [1, 0, 0, 0] # non-existing povar. return error,x,y,z

z0 = -0.5 * (b + math.sqrt(d)) / a
x0 = (a1 * z0 + b1) / dnm
y0 = (a2 * z0 + b2) / dnm

return [0, -y0, -x0, -z0]

# Inverse kinematics
# Helper functions, calculates angle theta1 (for YZ-plane)
def angle_yz(y0, x0, z0, theta=None):
    # y0 = -y0
    # x0 = -x0
    # z0 = -z0
    y1 = -0.5 * 0.57735 * f # f/2 * tg 30
    y0 -= 0.5 * 0.57735 * e # shift center to edge
    # z = a + b*y
    a = (x0 * x0 + y0 * y0 + z0 * z0 + rf * rf - re * re - y1 * y1) / (2.0 * z0)
    b = (y1 - y0) / z0

    # discriminant
    d = -(a + b * y1) * (a + b * y1) + rf * (b * b * rf + rf)
    if d < 0:
        return [1, 0] # non-existing povar. return error, theta

    yj = (y1 - a * b - math.sqrt(d)) / (b * b + 1) # choosing outer povar

```

```
zj = a + b * yj
theta = math.atan(-zj / (y1 - yj)) * 180.0 / pi + (180.0 if yj > y1
else 0.0)

return [0, theta] # return error, theta

def inverse(x0, y0, z0):
    # x0 = -y0
    # y0 = -x0
    # z0 = -z0
    theta1 = 0
    theta2 = 0
    theta3 = 0
    status = angle_yz(x0, y0, z0)

    if status[0] == 0:
        theta1 = status[1]
        status = angle_yz(x0 * cos120 + y0 * sin120,
                        y0 * cos120 - x0 * sin120,
                        z0,
                        theta2)
    if status[0] == 0:
        theta2 = status[1]
        status = angle_yz(x0 * cos120 - y0 * sin120,
                        y0 * cos120 + x0 * sin120,
                        z0,
                        theta3)
    theta3 = status[1]

    return [status[0], theta1, theta2, theta3]
```

4. Componentes del robot

En esta sección vamos a describir como se ha diseñado y construido el robot delta que ocupa este proyecto. Se explicarán las decisiones tomadas en cuanto a diseño de las piezas impresas con una impresora 3D, selección de piezas complementarias, estructura empleada y otros componentes. Se hablará también de los actuadores empleados y sus correspondientes controladoras, justificando su elección y explicando algunos de los problemas a los que se ha tenido que dar solución.

4.1. Diseño mecánico

Al tratarse de un robot con fines puramente didácticos y académicos se ha tratado de regirse más por disponibilidad de componentes, modularidad en la construcción (para facilitar la modificación de alguna característica, en caso de que sea necesario, sin aumentar el coste en exceso) y sencillez en el montaje.

Por esta razón se toman algunas decisiones iniciales:

- La estructura que soportará el robot será una estructura metálica de 310 *mm* de altura que se encontraba disponible en el laboratorio.
- Aprovechando la disponibilidad de una impresora 3D en el laboratorio se tratará de realizar el máximo número de piezas a través de esta técnica.

En segundo lugar se ha tratado de definir las dimensiones de los parámetros más importantes en la cinemática del robot y que marcarán la amplitud de su espacio de trabajo. Para ello se ha realizado un script de Matlab, aprovechando la cinemática descrita en la sección 3, que dibuje el espacio de trabajo del robot en función de sus parámetros para poder comparar las distintas configuraciones y elegir una que se ajuste a nuestros requerimientos.

Cabe señalar que el plano de trabajo se debe situar cerca del plano $z = 215 \text{ mm}$ ya que a la altura de la estructura hay que restar la altura de la base y los motores que se pondrán debajo y la de la pizarra que estará debajo.

El cálculo de cada espacio de trabajo es lento así que simplemente vamos a mostrar el espacio de trabajo con los parámetros de diseño definitivos (Figura 14).

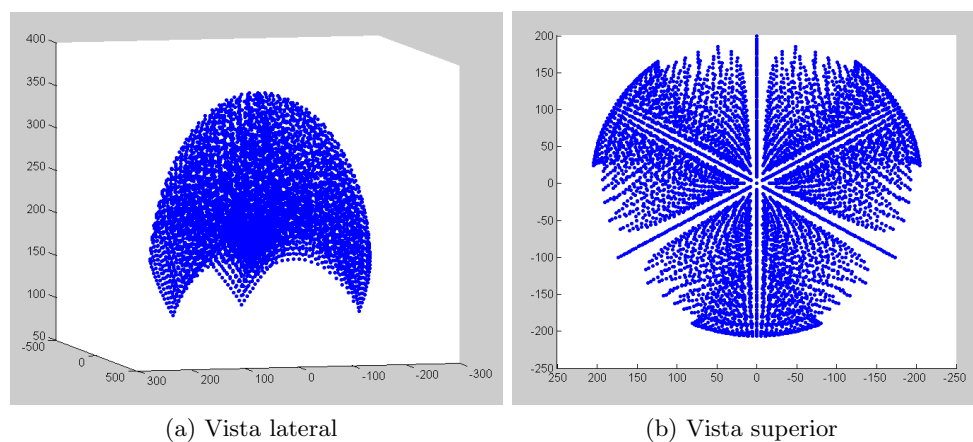


Figura 14: Espacio de trabajo representado a través de Matlab

Este espacio de trabajo tiene dos ventajas: tiene un brazo superior no demasiado grande (con el consiguiente ahorro de material al imprimir los brazos con la impresora 3D) y en su corte con el plano $z = 215 \text{ mm}$ corta una superficie suficiente como para cubrir toda la

pizarra que se usará como espacio de trabajo para el robot real. Así pues, los parámetros de diseño quedan del siguiente modo:

- Dimensión del triángulo superior que define la **base fija**: su lado será de **125 mm**. Limitado por la estructura metálica, ya que tiene esta base fija tiene que entrar en la plancha metálica de esta estructura.
- La longitud del **brazo superior** será de **150 mm**. Ajustado para ahorrar material en la impresión y para que el movimiento al final del brazo con cada paso de los motores no sea excesivamente grande.
- Los **brazos inferiores** tendrán una longitud de **205 mm**. Limitado por la altura de la estructura metálica y la posición donde se debe colocar el plano de trabajo.
- El lado del triángulo equilátero que representa la **base móvil** será de **130 mm**.

4.1.1. Diseño de piezas para impresión 3D

Como se ha comentado se trata de aprovechar al máximo la disponibilidad de recursos ya existentes en el laboratorio, es por ello que gran parte de las piezas serán realizadas mediante impresión 3D. La impresora disponible en el laboratorio y empleada en este proyecto es la BCN3D Sigma².

La BCN3D Sigma es una impresora 3D profesional de sistema FFF (*Fused Filament Fabrication*). Se caracteriza por su sistema IDEX (*Independent Dual Extruder*), que le permite imprimir sin limitaciones geométricas y ofrece la posibilidad de combinar técnicas existentes como impresiones multimateriales (con soportes) o multicolores.



Figura 15: BCN3D Sigma

Del mismo modo, y para no tener que pagar licencias de software para realizar diseños sencillos como los que nos ocupan se ha decidido emplear software libre. Todos los diseños de este proyecto se han realizado con FreeCAD.

FreeCAD es un modelador 3D paramétrico hecho inicialmente para diseñar objetos de la vida real de cualquier tamaño. El modelado paramétrico permite modificar fácilmente tu diseño regresando dentro del historial del modelo y cambiando sus parámetros. FreeCAD es de código abierto y altamente personalizable, programable mediante scripts y extensible. Es,

²Proyecto de la Fundació CIM, entidad adscrita a la UPC

además, multiplataforma (Windows, Mac y Linux), lee y escribe muchos formatos de archivos libres tales como STEP, IGES, STL, SVG, DXF, OBJ, IFC, DAE, etc.

Todas estas características lo hacen muy adecuado para diseñar piezas para la impresión 3D y para el prototipado ya que puedes modificar fácilmente y de forma ágil la geometría de una pieza si aprovechas bien el potencial del modelado paramétrico.

Vamos ahora a describir de forma somera las piezas que se han creado (Los planos en detalle se encuentran en el Anexo A):

- **Base superior:** Esta pieza se atornilla a la estructura a través del agujero central y posee agujeros para tornillos y tuercas M3 para colocar cada uno de los brazos en la posición definida. Se define en detalle en el plano 1.

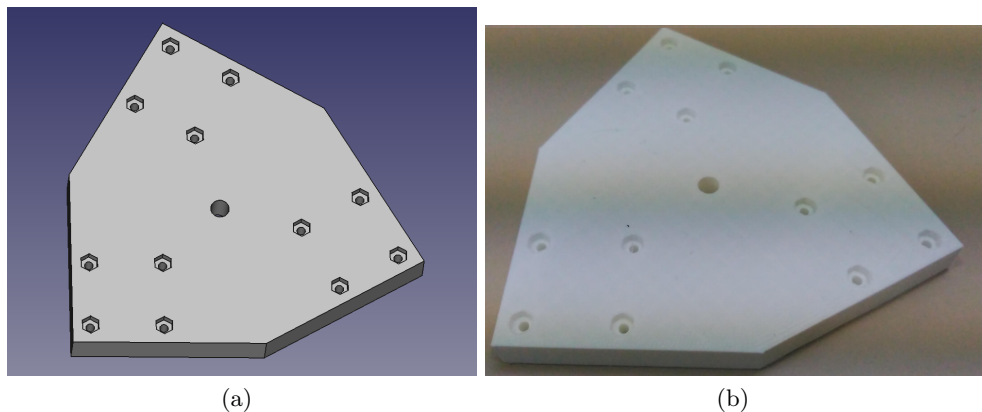


Figura 16: Diseño y pieza real de la base superior

- **Sujección para motores:** Se utilizan un total de tres piezas iguales a esta. Estas tres piezas se atornillan al soporte superior para facilitar la colocación de los motores. Tienen un ligero achaflanado para aumentar su rigidez y evitar que se deforme fácilmente con el calentamiento de los motores. Posee en una cara, en la que se atornilla el motor paso a paso seleccionado a través de cuatro agujeros de M3 (con un avellanado para alojar las cabezas de los tornillos sin que estos sobresalgan de la superficie de la cara e interfieran con el movimiento de los brazos), un agujero para adaptarse a la forma del motor y el agujero para el eje. Mientras que en la otra cara tiene otros cuatro agujeros de M3 con una cavidad para la cabeza de los tornillos que realizarán la unión con la base superior. Se define en detalle en el plano 2.
- **Brazo superior:** Se utilizan tres piezas como esta. Se unen a los ejes de los motores por un lado y a la estructura de barras que conforman los brazos inferiores por el otro. Cada una de las uniones posee un ajuste con tornillos de M3 consistente en un agujero pasante que llega hasta el agujero del eje o tubo (parte superior o inferior del brazo respectivamente) que atraviesa una cavidad de forma prismática en la que se aloja una tuerca que permite apretar el tornillo contra el eje o tubo, realizando un ajuste fuerte. Se define en detalle en el plano 3.
- **Base final:** base final móvil que porta el actuador (en nuestro caso un rotulador) en el centro y que esta unido al final de los tres brazos inferiores. Tiene un ajuste para las uniones con un tornillo de M3. El ajuste en este caso es más sencillo y simplemente son tres agujeros pasantes, uno en cada una de las cavidades tubulares que alojarán el final de los brazos inferiores, que permiten fijar con sendos tornillos M3 planos los tubos en la base final. Los tres salientes prismáticos harán la función también de fijación para

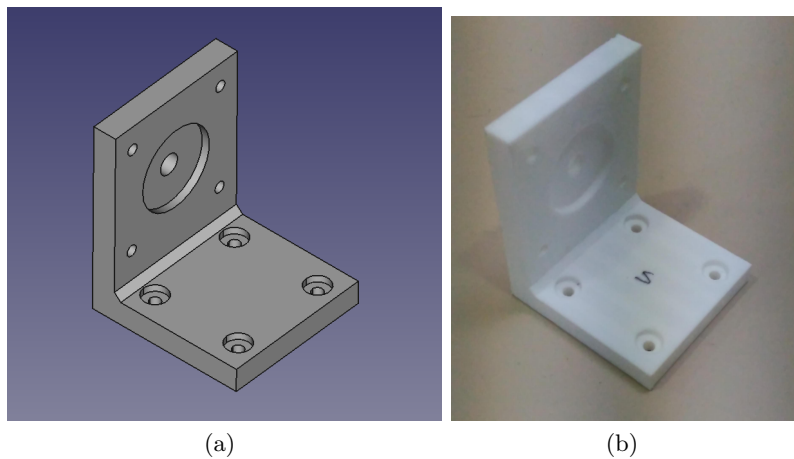


Figura 17: Diseño y pieza real de la sujección para motores

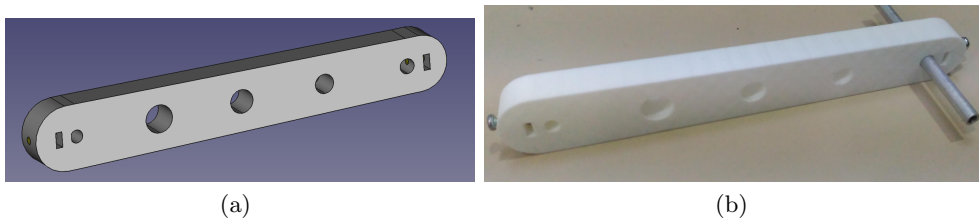


Figura 18: Diseño y pieza real de los brazos superiores

las gomas que se pondrán el soporte del actuador para mantener el rotulador pegado a la pizarra mientras se dibuja. Se define en detalle en el plano 4.

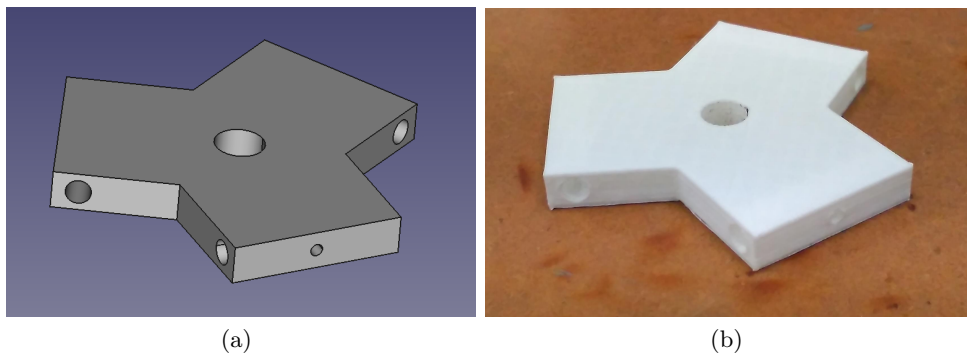


Figura 19: Diseño y pieza real de la base móvil

- **Soporte actuador:** pequeña pieza que se coloca en el cuerpo del actuador (rotulador) y que mantiene este presionado contra el plano de trabajo a través de gomas elásticas. Su agujero central aloja al rotulador haciendo tope, impidiendo al segundo salir hacia arriba, en la dirección perpendicular al plano y alejándose. Mientras que los tres salientes de los que disponen permiten colocar las gomas uniendo a la base final y al soporte del actuador en todo momento y provocando que el rotulador ejerza presión hacia abajo, en la dirección perpendicular a la pizarra y acercándose. Se define en detalle en el plano 5.

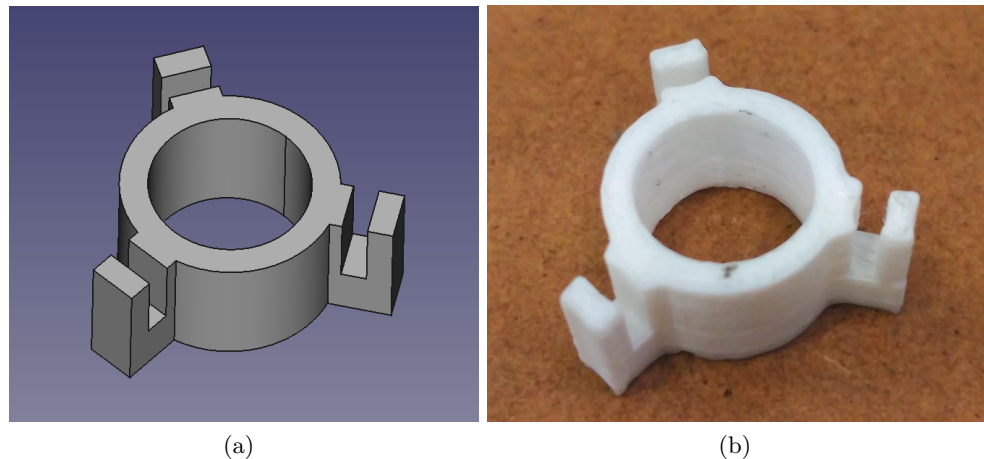


Figura 20: Diseño y pieza real de el soporte del actuador

- **Pinzas:** Piezas que se soportan a los brazos en la posición de origen. Dos tienen el voladizo que soporta al brazo hacia la derecha y otra hacia la izquierda para permitir colocar los drivers encima de la estructura. Su funcionamiento es sencillo, la parte superior con la hendidura se encaja en la plataforma metálica de la estructura mientras que el voladizo inferior está a la altura adecuada para soportar los brazos en la posición origen. Se define en detalle en el plano 6.

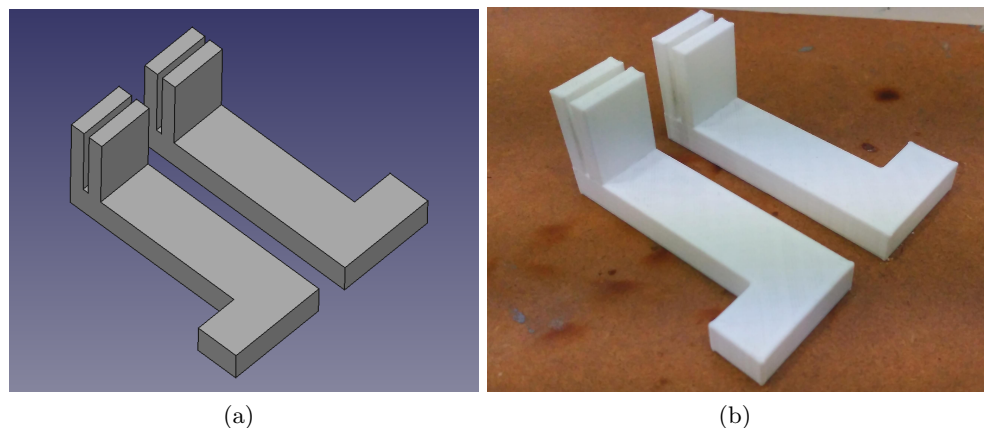


Figura 21: Diseño y pieza real de las pinzas

- **soporte drivers:** Esta pieza se sitúa sobre la estructura metálica para dar una ubicación estable para las controladoras (drivers). El agujero central deja acceso al tornillo que une la estructura metálica y la base superior, los dos vaciados de forma ovalada no tienen un finalidad funcional, sino que simplemente buscan reducir el material empleado para imprimir y reducir el tiempo de impresión y coste. Mientras que la función del recorte rectangular que existe en uno de sus lados largos es permitir situar la pinza en su posición. Las dimensiones están pensadas para colocar sobre ella las dos controladoras de mayores dimensiones enfrentadas por su lateral más largo y poder situar encima la controladora restante y el Arduino.

Con esto tenemos todas las piezas de plástico listas para realizar el montaje del robot.

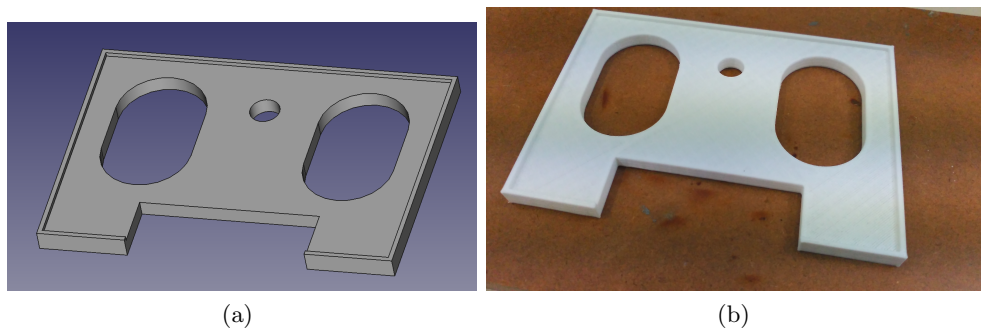


Figura 22: Diseño y pieza real del soporte para las controladoras

4.1.2. Otras piezas y componentes

Además de las piezas de plástico se emplean otras piezas y componentes metálicos en su mayoría disponibles en el laboratorio y que han sido reaprovechados para reducir el costo de realización del robot delta. A continuación enumeramos estos componentes igualmente necesarios para ensamblar nuestro robot:

- **Brazos inferiores:** Para construirlos se emplean cuatro segmentos de tubos de aluminio de sección circular, cuatro barras roscadas de M3, cuatro uniones de bolas y tuercas para todos los extremos. Los dos tubos más largos son atravesados por sendas barras roscadas y se cierra a ambos lados con la tuerca correspondiente y una unión de bolas. A la hora de montar el robot debemos de tener los componentes semi-ensamblados como se muestra en la figura 23a. Los otros dos tubos se ajustarán uno en el extremo final de los brazos superiores y otro en la cavidad destinada a este fin de la base móvil durante el montaje, para ser atravesados por las dos barras roscadas restantes y realizar la unión con las uniones de bolas. Los brazos montados fuera de su posición en el robot quedarían como se muestra en la figura 23b.

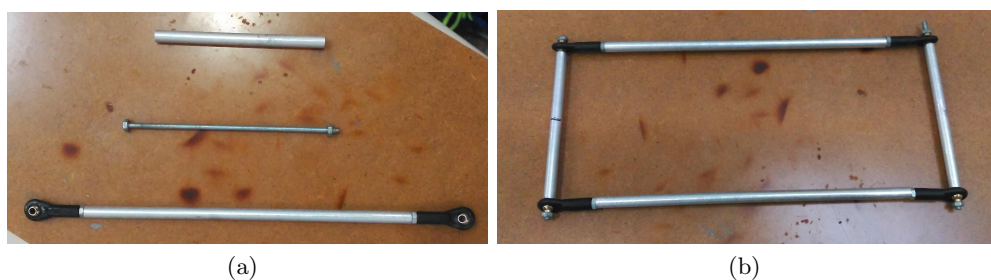


Figura 23: Brazo inferior montado y componentes

- **Estructura de soporte:** Una estructura conformada por una plancha triangular con tres patas roscadas soldadas que terminan en unas pequeñas zapatas de goma regulables. La plancha metálica se sostiene a una altura de 310 mm y posee un agujero M5 central para realizar la unión con la base superior. Se emplea esta estructura porque estaba disponible en el laboratorio y para ahorrar costes.
- **Plano de trabajo y actuador:** El plano de trabajo donde dibujará el robot será una pizarra blanca con marco de madera con unas dimensiones de 40x30 cm que se situará debajo de la base metálica. Mientras que el actuador será un rotulador BIC Velleda de



Figura 24: Estructura de soporte metálica para el robot

color negro adecuado para este tipo de pizarras y de fácil borrado. En la figura 25 se muestran ambos.



Figura 25: Pizarra y rotulador

- **tornillería, cableado y material menor:** Finalmente serán necesario tornillería variada (tornillos M3 planos y de diversas dimensiones con sus tuercas correspondientes, un tornillo M5 con dos arandelas y una tuerca adecuadas...), cables para realizar todas las conexiones lógicas y de alimentación, bridas para fijar los cables y mantenerlos ordenados, gomas elásticas para el actuador...

4.2. Motores paso a paso

Los motores paso a paso (*steppers* en inglés) son motores de corriente continua que se mueven en pasos discretos equidistantes. Estos motores tienen múltiples bobinas organizadas en grupos llamados "fases". Aplicando corriente a cada fase en una secuencia determinada, el motor girará un paso cada vez que se aplique un pulso de corriente a una fase.

Estos motores existen en muchos tamaños, estilos y características eléctricas. En cuanto a los tamaños NEMA³ define un standard relativo a la geometría básica de este tipo de motores, de modo que podemos encontrar entre la descripción de un motor las siglas NEMA acompañadas de un número, por ejemplo "NEMA 17" o "NEMA 23", que describen la geometría de la cara de montaje del motor, en concreto el número expresa el lado de esta cara en pulgadas si interponemos una coma entre ambos números, es decir un motor "NEMA 17" tendrá un lado de 1,7" [12]. En la figura 26 se muestran las dimensiones principales de los tamaños más populares, el largo de los motores, así como el de los ejes y la forma de los mismos no están especificados, además las aristas de estos motores suelen estar redondeadas o achaflanadas.

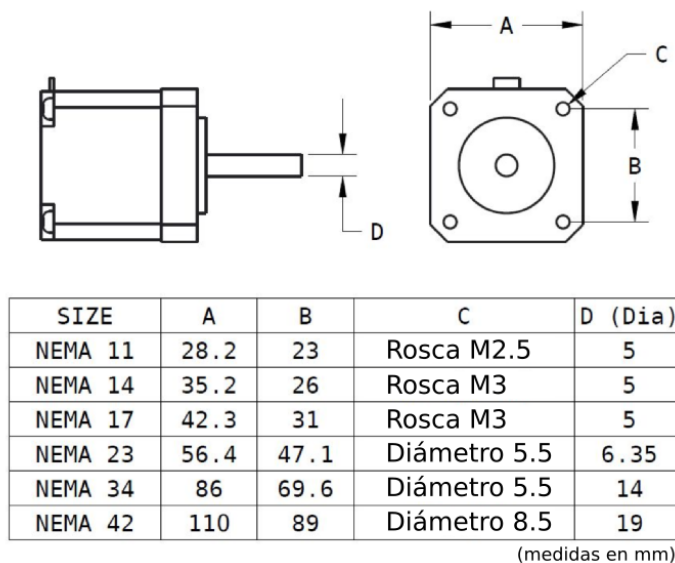


Figura 26: Tamaños más populares del standard NEMA

En cuanto a sus componentes y configuración, existen tres tipos fundamentales de motores paso a paso[13]

- Motor paso a paso de reluctancia variable: tienen un rotor de hierro puro y funcionan apoyándose en el principio de que la mínima reluctancia tiene lugar con la mínima distancia o intervalo espacial, por lo que los puntos del rotor son atraídos hacia los polos magnéticos del estátor. Suelen tener un paso de unos 15°.
- Motor paso a paso de rotor de imán permanente: usan un imán permanente en el rotor y operan basándose en la atracción y repulsión entre el imán permanente del rotor y los electroimanes del estátor. Se pueden obtener pasos con este motor que van desde los 7,5° hasta los 90°.
- Motor paso a paso híbrido: Esta configuración es una mezcla de las dos anteriores. Se caracteriza por tener varios dientes en el estátor y en el rotor y el rotor con un imán concéntrico magnetizado axialmente alrededor de su eje. Puede dar paso muy pequeños de hasta 1,8° o incluso 0,9°.

Por otro lado, existen dos tipos básico de disposiciones para los devanados de las bobinas, distinguiéndose motores unipolares y bipolares:

³National Electrical Manufacturers Association, <http://www.nema.org/>

- Motores paso a paso unipolares: tienen un devanado con una toma central para cada fase, de forma que cada sección del devanado alimentada para cada dirección del campo magnético. Dado que en esta disposición un polo magnético puede ser invertido sin cambiar el sentido de la corriente, el circuito de comunicación puede ser muy simple (un simple transistor para cada devanado). Dependiendo de la conexión interna suelen tener 5 o 6 cables de salida, tendrán 6 cables cuando cada par de bobinas tenga el común separado, mientras que tendrá 5 cables si las cuatro bobinas tienen un polo común.
- Motores paso a paso bipolares: poseen un único devanado por fase. La corriente en un devanado debe ser invertida cuando se quiera invertir el polo magnético, por lo que el circuito de conducción de estos motores será más complicado, habitualmente empleando una configuración tipo puente-H. Existen dos cables de salida por fase y ninguno de ellos común, por lo que en general tienen 4 cables.

Tras entender un poco que es un motor paso a paso cabe preguntarse para qué sirve y para que no. Los motores paso a paso son buenos para tres tareas principalmente[14]

- Posicionamiento: dado que se mueven en pasos (precisos, repetibles y siempre iguales) los motores paso a paso destacan en aplicaciones que requieren un posicionamiento preciso, tales como impresión 3D, CNC, plataformas para cámara o Plotters, incluso algunos discos duros usan estos motores para posicionar la cabeza de lectura y escritura.
- Control de velocidad: De igual modo son muy útiles en aplicaciones que requieran un alto control sobre la velocidad angular como los procesos de automoción o la robótica.
- Alto par a baja velocidad: Los motores de corriente continua habituales no tiene demasiado par a bajas velocidades, en contraposición un motor paso a paso entrega su máximo par a velocidades reducidas.

Sin embargo, este tipo de motores también tienen limitaciones, a destacar:

- Baja eficiencia energética: En los motores paso a paso el consumo de corriente es independiente de la carga, en contraposición con los motores de corriente continua habituales. Además consumen la máxima corriente incluso cuando no están moviéndose, es por esta razón que tienden a calentarse bastante.
- Par a alta velocidad limitado: En general, los motores paso a paso tienen menos par a altas velocidades, si bien es cierto que existen motores paso a paso optimizados para una mejor actuación a altas velocidades, aunque necesitan de *drivers* apropiados.
- Lazo abierto: Al contrario que los servo-motores, la gran parte de los motores paso a paso trabajan en lazo abierto, sin información ni realimentación sobre la posición actual, por lo que no detectará si se salta o pierde un paso.

4.2.1. Motor paso a paso empleado

Para nuestro robot no se necesitan motores excesivamente grandes (obligaría a tener una base muy grande), pesados (la base tendría que ser más robusta lo que implicaría también un mayor uso de material y coste) ni potentes (el objetivo final es desplazar un rotulador sobre una pizarra horizontal) pero sí con un paso pequeño (para tener mayor precisión).

El modelo de **motor seleccionado** para este proyecto es: Motor paso a paso Sanyo Denki Sanmotion, serie 103H5. Se trata de un motor paso a paso NEMA 17 de tipo híbrido, bobinado bipolar (4 cables), 200 pasos (1,8° por paso). Sus características principales se resumen en la tabla 1 y en la figura 27 podemos ver una imagen del mismo.

Característica	Valor
Tipo motor	Paso a paso híbrido
Colocación del Bobinado	Bipolar
Ángulo por paso	$1,8^{\circ}$
Par de sujección	$0,39Nm$
Número de cables	4
Tensión Nominal	$24V(DC)$
Tamaño bastidor	$42 \times 42mm$ (NEMA 17)
Profundidad	$40,5mm$
Diámetro del eje	$5mm$
Longitud del eje	$22,5mm$
Resistencia por fase	$4,1\Omega$

Cuadro 1: Características motor seleccionado

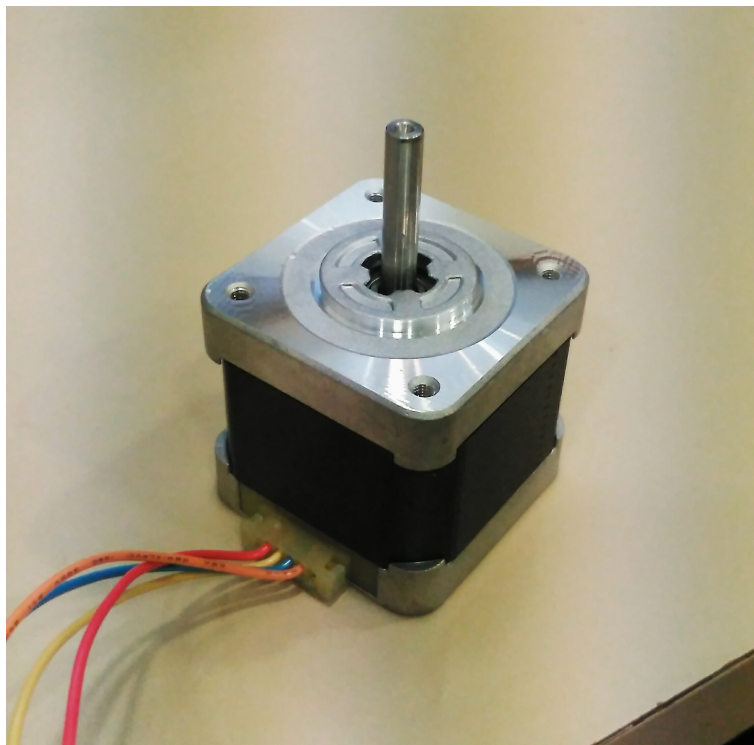


Figura 27: Motor paso a paso seleccionado

4.3. Controladores de los motores

Para hacer funcionar un motor paso a paso es necesario utilizar un chip controlador específico (*driver* en inglés) para este tipo de motores junto con un controlador para generar las señales lógicas o un microcontrolador con uno o dos chips con un puente-H completo integrados[15].

4.3.1. Opciones posibles

La primera opción mantiene la alimentación de los motores separada de la lógica (Arduino o cualquier otro controlador), debido a que un microcontrolador por si solo no puede suministrar suficiente potencia para mover los motores paso a paso directamente.

El controlador debe manejar tres señales, habitualmente se etiquetan en los circuitos controladores como "step" (o "PUL"), "DIR" y "GND". Estas señales llevan la información del movimiento que se desea realizar con el motor paso a paso. Este controlador es un dispositivo de lógica digital pura en general y requiere una alimentación pequeña ($\pm 5\text{ V}$).

El circuito controlador se conecta a los 4 cables del stepper motor (para un motor bipolar) y en su interior se alojan entre otras cosas grandes transistores de potencia. Necesitan por tanto una alimentación mayor y que sea suficiente para poder mover los motores, aunque es verdad que en algunos casos, algún motor paso a paso realmente pequeño puede ser movido directamente desde el controlador pero no es lo habitual..

Otra ventaja que tienen los chips controladores específicos es que pueden realizar *microstepping*, esto es que son capaces de realizar *fractional steps*, o dicho de otra forma, son capaces de reducir el tamaño de cada paso natural de motor en fracciones de $1/2$, $1/4$, $1/8$... y hasta $1/128$ en algunos casos. De esta forma podemos aumentar la resolución de nuestro motor sin cambiar el mismo.

Por otro lado, la segunda opción, microcontroladores con chips tipo puente-H integrados pueden conseguir altas velocidades de rotación en motores paso a paso. Con esta opción, es posible tener un extremo control sobre el momento exacto en que cada devanado es activado en el interior del motor. Esto es crucial para obtener altas velocidades de giro porque a medida que se incrementa la velocidad la sincronización en la activación de las bobinas tiene que ser perfecta.

Los puentes-H son necesarios para que el microcontrolador pueda suministrar la corriente necesaria a los motores, que es más de la que el microprocesador por si solo puede suministrar. Lo habitual es que un puente-H se emplee para controlar un motor de corriente continua corriente, pero en este caso, los chips con puente-H se utilizan para controlar con precisión la cantidad de electricidad que atraviesa cada bobina del motor paso a paso. En consecuencia, para motores paso a paso bipolares, se necesitaran dos chips para controlar cada motor.

4.3.2. Primera selección

En un primer momento se seleccionó un microcontrolador que incorporase varios puentes-H para controlar el robot delta de este proyecto, se optó por dos Adafruit Motor Shield V2 (figura 28a) completamente apilables con Arduino. De esta forma, se pretendía obtener un diseño más compacto y aprovechar que se trataba de chips adaptados a Arduino y con gran soporte.

Este *shield* monta dos chips TB6612 con dos puentes-H cada uno diseñados a partir de MOSFET con una capacidad de corriente de $1,2\text{ A}$ por canal y con capacidad para suministrar picos de 3 A durante 20 ms [16]. Están diseñados para producir caídas de tensión mínimas desde la tensión suministrada por la fuente o baterías, y además tienen los diodos de retorno incorporados. De esta manera, cada Adafruit Shield puede controlar 4 motores DC o 2 motores

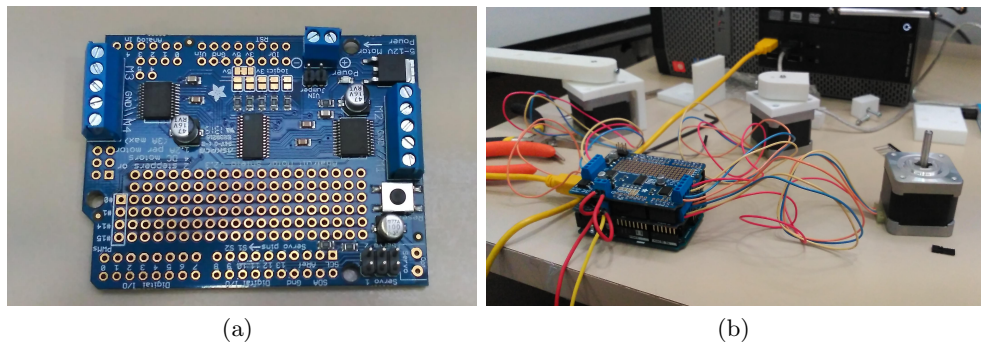


Figura 28: Adafruit Motor Shield V2 y montaje

paso a paso (unipolares o bipolares) con una tensión de alimentación variable entre 4,5 VDC y 13,5 VDC .

La alimentación se puede realizar directamente al Adafruit Shield o aprovechando la alimentación del Arduino al que este acoplado, ya que dispone de un *jumper* para este propósito. La lógica puede funcionar a 5 V o 3,3 V, lo cual es configurable a través de otro *jumper*.

En lugar de usar los pines con PWM del Arduino, esta placa tiene incorporado su propio chip totalmente dedicado a generar PWM. Las comunicaciones de control se hacen con I2C, de modo que solo dos pines (SDA y SCL) son necesarios para controlar múltiples motores. Esto lo hace totalmente compatible con cualquier Arduino y totalmente apilable con más Adafruit Shields del mismo tipo, poseyendo 5 bits para la selección de dirección significa que puede apilar hasta 32 shields (en otras palabras, 64 motores paso a paso o 128 motores DC comunes).

Dadas todas estas características se consideró este Adafruit Shield una buena opción con la que llevar a cabo el control de los motores paso a paso de este proyecto. En la figura 28b podemos ver el montaje de Arduino UNO con dos Adafruit Shield apilados moviendo los tres motores paso a paso del proyecto.

Sin embargo, finalmente se desechó la idea de continuar utilizando esta opción principalmente por los siguientes motivos:

- Exceso de calentamiento: los motores paso a paso se calentaban rápidamente desprendiendo gran calor que se transmitía por todo el cuerpo del motor hasta el eje y de este a los brazos superiores del robot y reblandecía el plástico del que están formados por lo que llegaba a deformarlos. Pese a que el Adafruit Shield está diseñado para entregar 1,2 A por alguna razón estaba trabajando gran parte del tiempo por encima de este valor, lo que generaba que los motores recibieran más corriente que su corriente nominal y todo este exceso se transformase en pérdidas de calor.
- Mal *microstepping*: A pesar de que las librerías de Adafruit incluían *microstepping* como una opción para controlar los motores, estas no funcionaban todo lo bien que cabía esperar y había demasiada pérdida de par en algunas ocasiones.
- *Microstepping* insuficiente: también es importante decir que solo se podía conseguir la mitad del paso habitual del motor, es decir pasar de un paso de $1,8^\circ$ a uno de $0,9^\circ$, lo cual resultó insuficiente resolución ya en las primeras pruebas.

4.3.3. Segunda selección

Por lo explicado en la sección anterior, se buscó una alternativa para el control de los motores paso a paso del robot delta. Se decidió finalmente emplear chips controladores específicos



(a)

(b)

Figura 29: Controlador HY-DIV268N-5A

y manejar las señales de control a través de Arduino UNO.

En el laboratorio se disponía dos controladores específicos para motor paso a paso bifásicos HY-DIV268N-5A [17] provenientes de algún proyecto anterior y que se decidió reutilizar.

Se trata de controladores específicos con *microstepping* de alto rendimiento basados en el circuito integrado TOSHIBA TB660HG. Este circuito integrado utiliza un único chip regulador de intensidad con PWM sinusoidal y para motores paso a paso bipolares. Los pulsos sinusoidales favorecen una menor vibración y una mayor eficiencia al mover el motor. El HY-DIV268N-5A puede mover motores paso a paso de 2 o 4 fases desde NEMA 17 hasta NEMA 34 siempre y cuando tengan una intensidad nominal entre 0,2 A y 5 A.

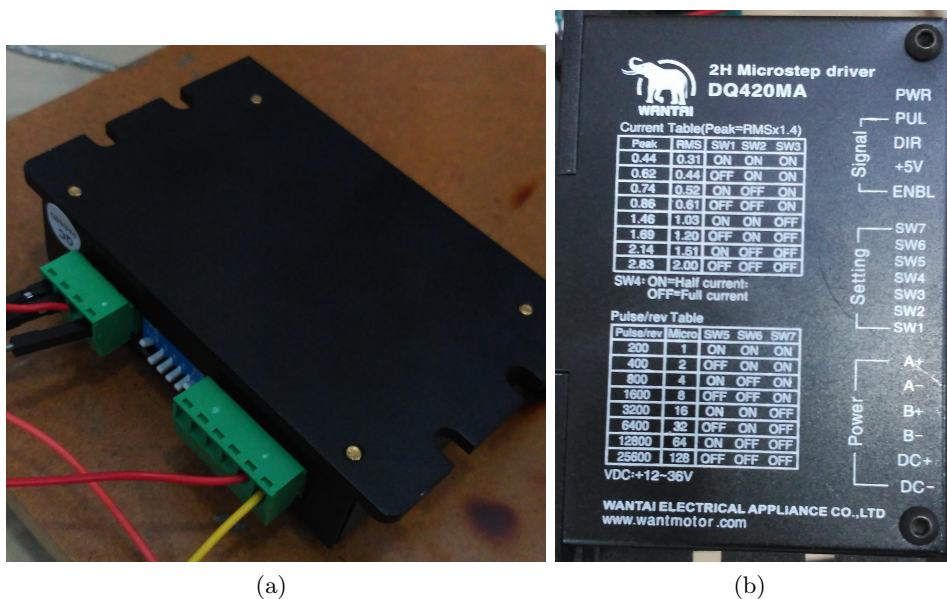
Trabaja con tensiones de alimentación de hasta 50 V. Permite trabajar con *microstepping*, el cual es ajustable, de 1/2, 1/4, 1/8 y 1/16. Posee protecciones de sobrecarga, sobrecalentamiento, sobrealimentación y infraalimentación para evitar que se dañe el dispositivo en cualquiera de estos escenarios. El controlador viene montado en una caja con un sistema de enfriamiento de aluminio que protege además el circuito de suciedad, líquidos, polvo...

Pero dado que son tres motores a controlar en nuestro robot, era preciso adquirir un controlador más y, finalmente, se optó por el controlador DQ420MA [18] que es compatible con los dos controladores con los que ya se contaba y estaba más adaptado al tamaño de los motores NEMA 17 de 1A que se quería controlar.

Se trata también de un controlador específico para motores paso a paso de dos fases híbridos, con capacidad para funcionar con tensiones de alimentación de entre 12 VDC y 36 VDC, menos de 2 A por fase del motor. Posee también protecciones de sobrealimentación, infraalimentación, sobrecarga y corto circuito de fases. Es capaz de generar pulsos de *microstepping* desde fracciones de 1/2 y 1/4 hasta 1/128 con gran rendimiento.

Los dos controladores funcionan de forma similar en cuanto a las conexiones y a su ajuste. Básicamente, se distinguen dos bloques de conexiones y una serie de interruptores para la configuración:

- Bloque de conexiones de **alimentación y potencia**: posee seis entradas, dos son la alimentación ($DC+$ y $DC-$), y los otros cuatro los cables del motor paso a paso bifásico ($A+$, $A-$, $B+$ y $B-$) dos para cada bobinado.
- Bloque de conexiones para las **señales de control**: Aquí conectaremos las tres señales de control que enviará Arduino a cada controlador. EN o $ENBL$ para habilitar el giro libre del motor o bloquearlo, DIR para seleccionar la dirección de avance y PUL para enviar los pulsos que excitaran el motor y le harán avanzar un paso o un "micropaso" si



(a)

(b)

Figura 30: Controlador DQ420MA

se emplea *microstepping*. En el controlador HY-DIV268N-5A se tiene libertad para conectar cada señal de control descrita, mientras que en el DQ420MA existe una toma común etiquetada con +5V donde se debe conectar la referencia de voltaje del circuito lógico, así para activar *PUL*, *ENBL* o *DIR* tendremos que enviar una señal de 0 V y cuando queramos desactivarlas una de 5 V (o 3,3 V si es el caso).

- **Bloque de interruptores para configuración:** Colocando los interruptores en la posiciones indicadas en las tablas que se encuentran en el reverso de los controladores (figuras 29b y 30b) podemos regular la corriente de alimentación y si queremos usar los pasos fraccionados y con que fracción.

Se usará una fuente de alimentación de 12 VDC común para los tres controladores, se alimentarán los motores a 1,2 A y se usarán *microsteps* que sean comunes a ambos controladores.

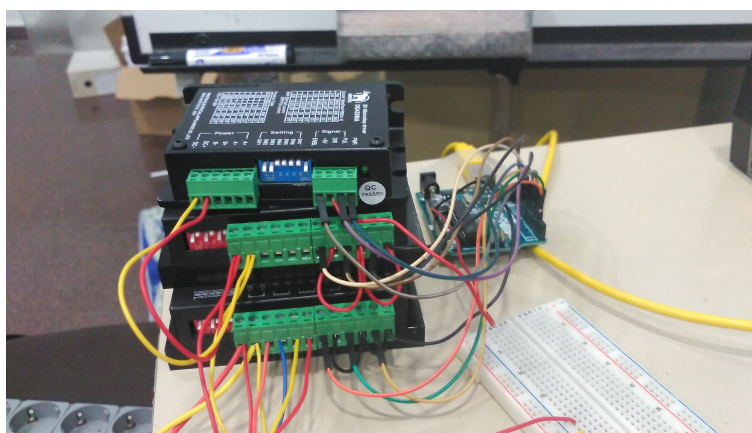


Figura 31: Pila de controladores conectados a fuente de alimentación de 12 V

En la figura 32 podemos ver el esquema de las conexiones entre Arduino, controladoras y motores paso a paso.

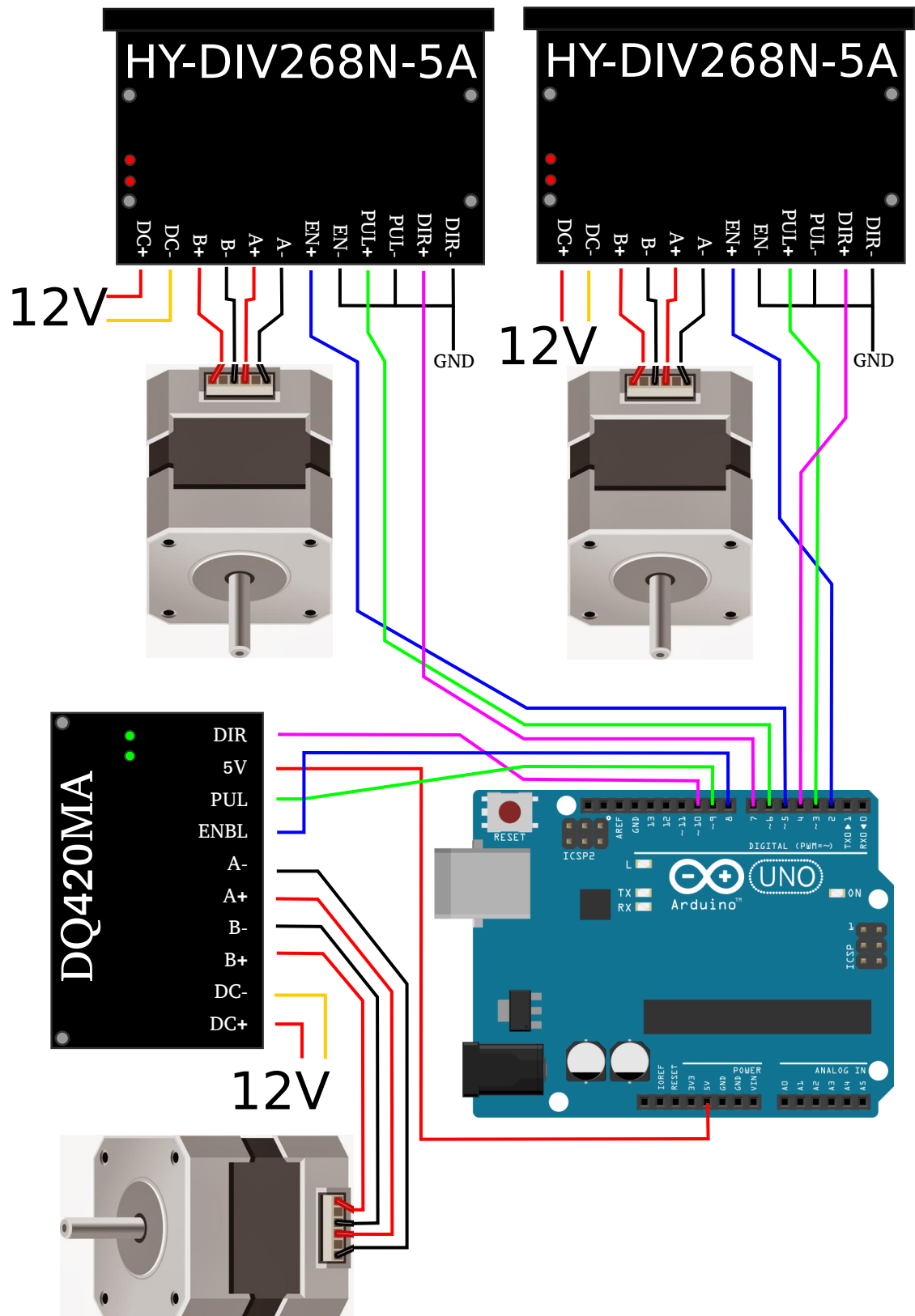


Figura 32: Esquema de conexiones

4.4. Montaje

Una vez se han descrito todos los componentes del robot, pasamos a describir a grandes rasgos el ensamblaje de todas las partes para dar lugar al robot final.

En primer lugar se atornillan las sujeciones de los motores a la base superior con cuatro tornillos M3 de forma que nos queda un bloque como el de la figura 33.

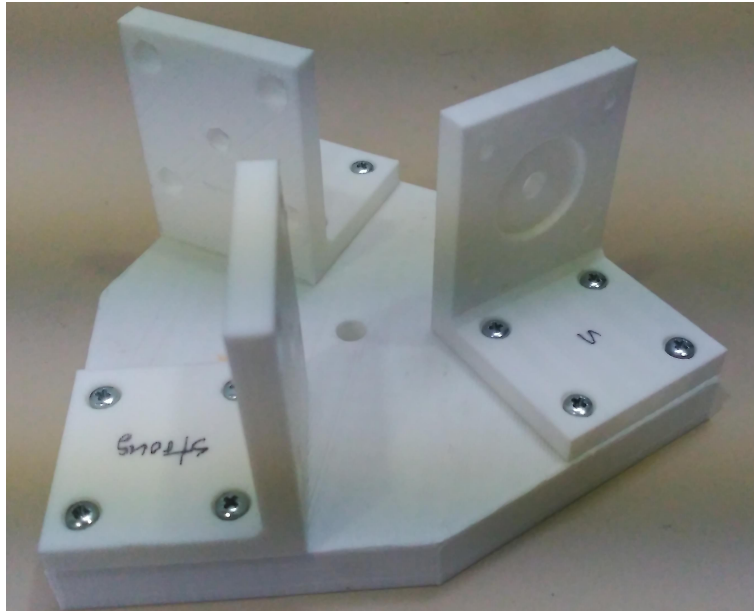


Figura 33: Base superior y sujeciones para los motores acoplados

A continuación, se colocan los motores uno a uno en cada una de las sujeciones colocando una espuma en la parte superior para amortiguar las vibraciones que puedan existir con el robot en movimiento, y situando los cables del motor hacia el exterior de la base superior. La unión se realizará con cuatro tornillos M3 planos por motor y sin usar tuercas. El resultado se muestra en la figura 34.

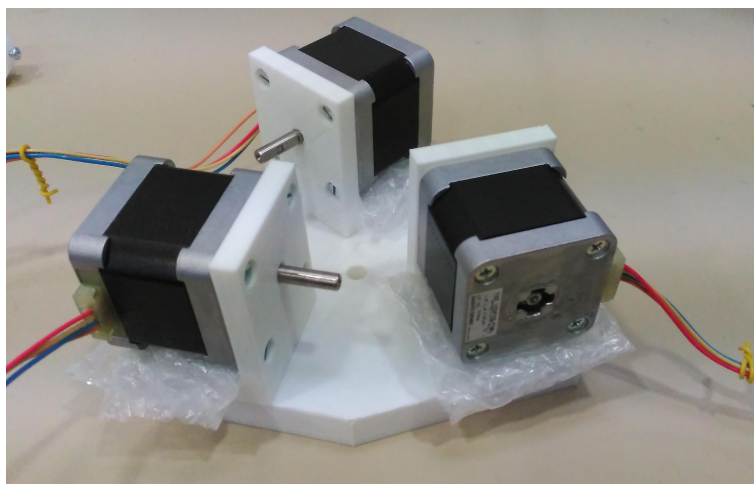


Figura 34: Motores paso a paso colocados en su posición

Colocamos ahora los brazos superiores en el eje de cada motor y apretamos con el ajuste diseñado a través de tornillo y tuerca M3 hasta que la unión sea fuerte pero no tanto como para deformar la pieza de plástico. Y ya podemos colocar todo el bloque en la estructura metálica

que soportará el robot. Esta unión se realiza a través de un conjunto tornillo, arandelas y tuerca de M5. Podemos ver este paso intermedio en la figura 35.

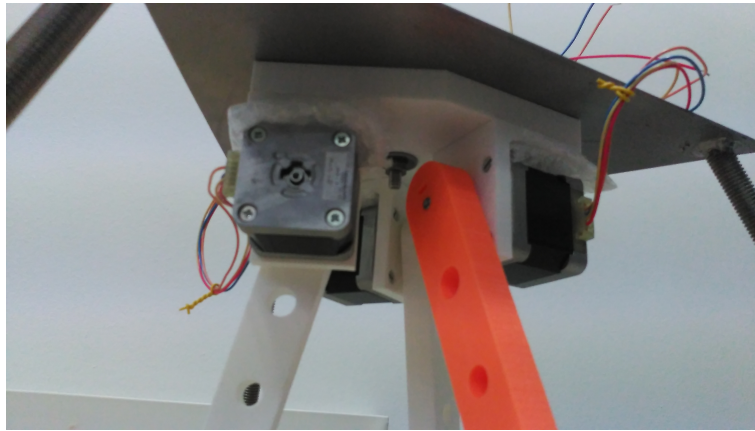


Figura 35: Base fija unida a la estructura metálica y brazos superiores unidos a los ejes

Es momento de colocar los brazos inferiores, para empezar se coloca en cada uno de los extremos de los brazos superiores uno de los tubos metálico pequeños bien centrado y se fijará a través de un tornillo y una tuerca M3. Después atravesamos cada uno de estos tubos con los cilindros roscados correspondientes junto con las dos barras laterales montadas con sus articulaciones.

Paralelamente hemos preparados la base inferior con los otros tres tubos pequeños, las gomas y el actuador, tal y como se muestra en la figura 36. Cerramos cadena cinemática pasando los cilindros roscados por las articulaciones finales de los brazos y por los tubos que hemos colocado en la base inferior.

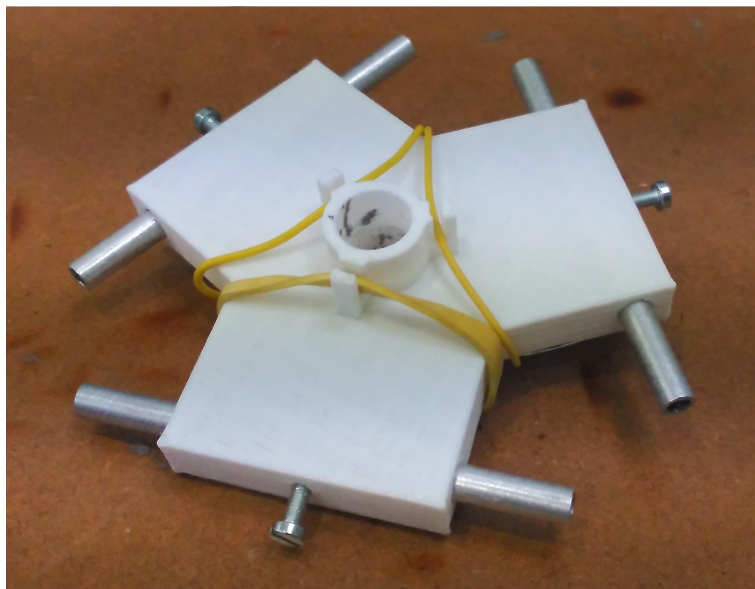


Figura 36: Actuador final con los tubos para la unión a los brazos inferiores y el soporte para el actuador colocados

Ahora hay que colocar las controladoras de los motores y el Arduino en su soporte, fijándolos con un poco de silicona, y colocar el conjunto sobre la estructura metálica (ver la figura 37). En este momento podemos conectar todos los motores convenientemente a su controladora y estos al Arduino y a la alimentación de 12 V.

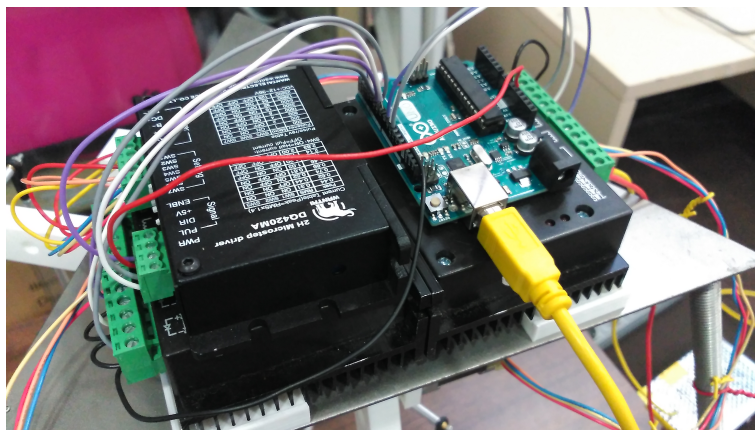


Figura 37: Controladoras y arduino sobre su soporte, en posición y con todas las conexiones realizadas

Finalmente, solo nos falta colocar el robot en su posición con la pizarra debajo, y fijar los brazos a la posición inicial a través de las pinzas. El robot está listo para dibujar (Figura 38).

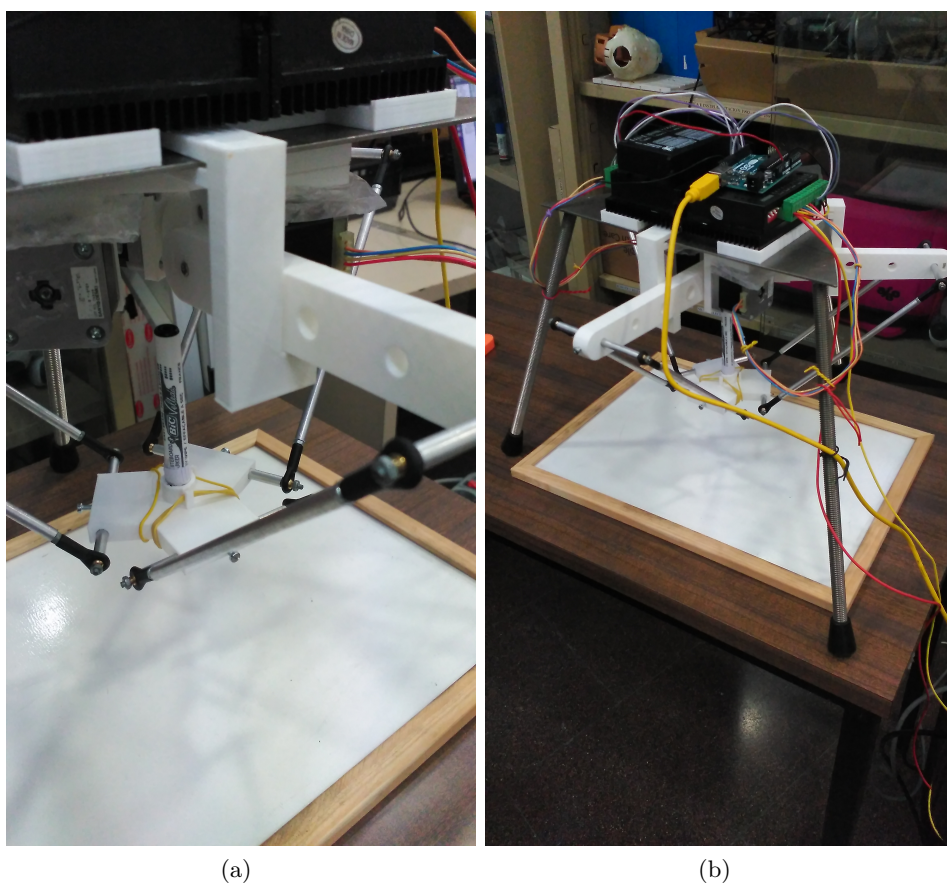


Figura 38: Montaje completo y robot en posición inicial

5. Creación de trayectorias

En esta sección vamos a describir el procedimiento mediante el cual se generan las trayectorias del robot y como se ejecuta su movimiento. En este proceso podemos distinguir cuatro grandes bloques:

- **La preparación y tratamiento de la imagen:** Se debe simplificar la imagen con un binarizado y convertirla a un formato conveniente.
- **La extracción de la información de la imagen:** Con la imagen lista es necesario extraer la información que contiene para generar los trazos del dibujo.
- **La generación de toda la trayectoria que tiene que seguir el robot:** Se genera todo el movimiento que va a realizar el robot, desde su acercamiento desde el punto de origen hasta su salida pasando por los trazos sobre el plano de trabajo y los movimientos entre trazos.
- **La conversión de esta trayectoria:** Toda la trayectoria debe ser traducida en algo que pueda ser ejecutado por Arduino y que transforme la trayectoria en movimiento de los actuadores y en consecuencia del robot. Este "algo" son trenes de pulsos, que en la sección 6 explicaremos como se ejecutarán.

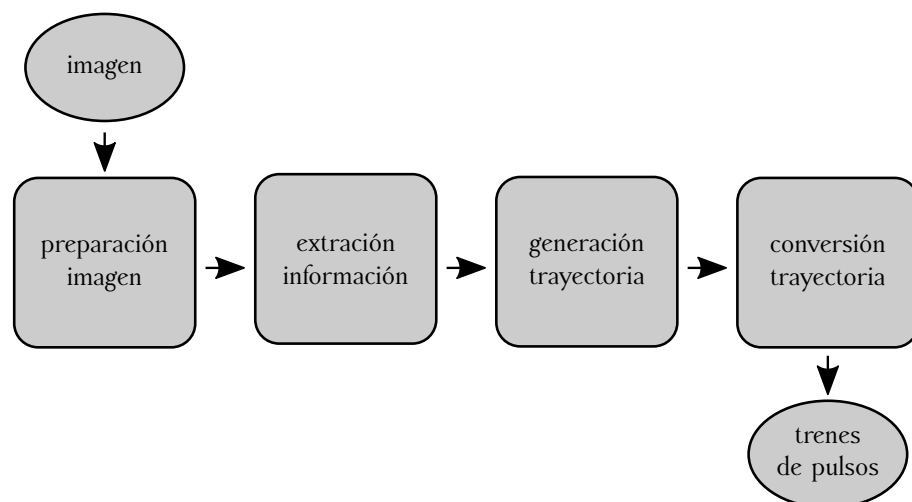


Figura 39: Diagrama de flujo del algoritmo de generación de trayectorias

5.1. Tratamiento de imágenes

Las imágenes que se desean dibujar pueden encontrarse en diversos formatos y en diversas formas y pueden no ser los adecuados para obtener la información de ella y extraer los recorridos más importantes de ellas. Por ello, es necesario realizar una serie de operaciones previas a la imagen para adaptarla convenientemente para ser empleada en las etapas posteriores.

Una imagen adecuada para realizar la siguiente etapa será una imagen en blanco y negro, es decir totalmente binarizada, y en formato SVG, que es un formato vectorial en donde no hay píxeles si no formas descritas por curvas y figuras geométricas.

Dado que queda fuera del alcance de este proyecto diseñar un método eficiente para aislar las formas de interés que existan en una imagen, se aplica un método sencillo aprovechándose de las herramientas que nos proporciona el software libre ImageMagick. Sencillamente, se empleará una de sus herramientas de línea de comandos, *convert*, que es capaz no solo de

convertir el formato de una imagen si no también de aplicar filtros, convolucionar máscaras sobre una imagen, realizar operaciones morfológicas... y mucho más.

Así se propone realizar las siguientes operaciones sobre una imagen:

- Una primera binarización convertirá la imagen a blanco y negro. Es importante ajustar bien el umbral para destacar las formas más importantes de la cara y no ocultarlas. Vemos la imagen original y la primera binarización en las Figuras 40a y 40b.

```
$ convert img_entrada.jpg -threshold 45% img_salida.jpg
```

- Podemos ver que tras la binarización existen muchos "huecos" en blanco en las formas que no sería posible representar en nuestro dibujo con el robot, por lo que se tratarán de eliminar. Con este fin se aplica un filtro gaussiano a la imagen y dado que esto provoca la aparición de niveles de gris en la imagen es necesario realizar un segundo binarizado. El resultado tras estas dos operaciones se muestra en la Figura 40c.

```
$ convert img_entrada.jpg -gaussian-blur 5x2 img_salida.jpg
```

```
$ convert img_entrada.jpg -threshold 45% img_salida.jpg
```

En el comando empleado para aplicar el filtro gaussiano se especifican dos valores, el primero de ellos (en este caso se ha fijado en 5) es el tamaño de la máscara gaussiana y el segundo (fijado en 2) es el valor que toma σ en la ecuación 14 que determinará lo fuerte que será el "difuminado" que provocará la aplicación del filtro.

$$G(u, v) = \frac{1}{2\pi\sigma^2} e^{-(u^2+v^2)/(2\sigma^2)} \quad (14)$$



Figura 40: Binarización y filtro Gaussiano sobre imagen

- Con el objetivo de simplificar aún más las formas para obtener una imagen con trazos realizables por el robot se aplicó una operación morfológica típica, un "cerrado". Esta operación consiste en una concatenación de dos operaciones morfológicas más sencillas, una dilatación y una erosión, ambas con la misma máscara (Figura 41). Con estas operaciones se suaviza la parte exterior de las figuras, llenando los agujeros que pudiesen existir en estas y uniendo figuras que estén muy próximas.

```
$ convert img_entrada.jpg -morphology Close Disk:4.3 img_salida.jpg
```

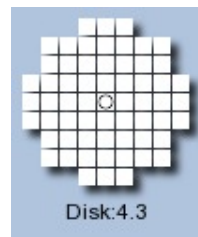


Figura 41: Máscara para operación de cerrado

- Si se deseara se podría extraer el contorno de las formas obtenidas con operaciones morfológicas. Simplemente realizando una operación de erosión a la imagen original y restando el resultado a la propia imagen original obtenemos los bordes como se muestra en la figura 43b. En este caso se usa una máscara más pequeña (Figura 42) para que los bordes sean finos.

```
$ convert c3.jpg -morphology EdgeIn Octagon:2 -negate img.salida.jpg
```



Figura 42: Máscara para operación de extracción de bordes

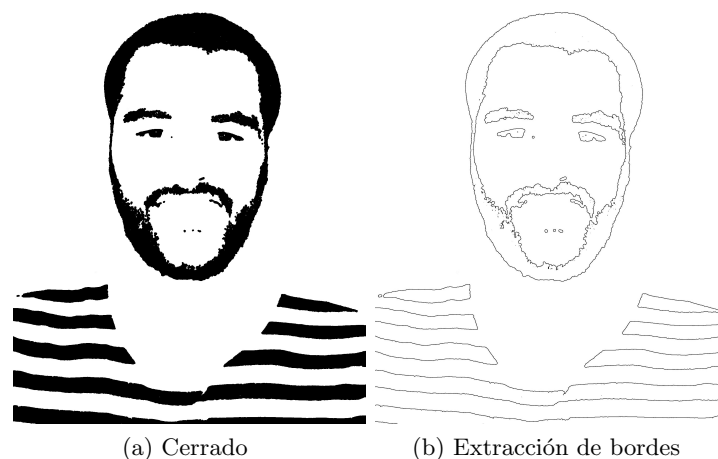


Figura 43: Operaciones morfológicas sobre imagen

Una vez la imagen está lista, falta convertirla al formato deseado, SVG. Para realizar esta conversión utilizaremos otra herramienta de software libre, Potrace. Potrace es una herramienta para transformar una imagen en forma de mapa de bits (como son los formatos PBM, PGM, PPM o BMP) en una imagen más suave y escalable como son las imágenes definidas con vectores (formatos EPS, PDF y SVG entre otros).

De este modo tenemos que realizar dos conversiones, primero convertir nuestra imagen a un formato aceptado por Potrace, utilizaremos por ejemplo el formato BMP, y en segundo

lugar convertirla a un formato vectorial con Potrace, como hemos dicho en este proyecto se empleará el formato SVG. Esto se realiza con dos sencillos comandos en nuestro terminal Linux:

```
$ convert img_entrada.jpg img_salida.bmp
$ potrace -s img_salida.bmp
```

Con lo que ya tenemos lista nuestra imagen para pasar a los siguientes pasos.

5.2. Extracción de curvas de Bézier

Una vez se ha conseguido una imagen en formato SVG adecuada, es necesario extraer la información que contiene. El formato SVG (*Scalable Vector Graphics*) define objetos vectoriales bidimensionales en un formato de texto plano tipo XML, los cuales no pierden calidad cuando son reescalados o ampliados. Este formato permite elementos geométricos vectoriales (rectas, curvas, áreas...), imágenes de mapa de bits/digitales y texto.

Los elementos geométricos en SVG son objetos con atributos genéricos, algunos de ellos básicos y otros optativos (que tienen un valor por defecto si son obviados). Todos los elementos de un archivo SVG se encuentran encerrados dentro de una ventana de dimensiones establecidas al comienzo del archivo (*width* y *height*). Además esta ventana tiene un sistema de coordenadas en su interior para situar todos los elementos del dibujo: el origen del sistema se coloca en la parte superior izquierda de la ventana, las coordenadas x positivas van hacia la derecha y las y positivas hacia abajo. Este sistema de coordenadas se puede transformar (escalar, mover...).

Listado de código 3 Ejemplo de pequeño archivo SVG generado con Potrace

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg version="1.0" xmlns="http://www.w3.org/2000/svg"
width="1920.000000pt" height="1080.000000pt" viewBox="0 0 1920.000000 1080.000000"
preserveAspectRatio="xMidYMid meet">
  <metadata>
    Created by potrace 1.11, written by Peter Selinger 2001-2013
  </metadata>
  <g transform="translate(0.000000,1080.000000) scale(0.100000,-0.100000)"
fill="#000000" stroke="none">
    <path d="M9421 10783 c3 -20 -38 -93 -52 -93 -5 0 -9 -10 -9 -22 0 -20 -1 -20
-18 -5 -21 19 -32 13 -32 -15 0 -10 -7 -18 -15 -18 -8 0 -15 -8 -15 -19 0 -10
-10 -22 -22 -26 -57 -19 -62 -25 -21 -20 73 9 96 22 90 51 -4 23 -2 25 19 21
13 -2 21 -1 19 3 -3 5 11 17 30 29 19 11 40 32 45 45 5 14 21 32 35 42 39 25
32 44 -16 44 -33 0 -40 -3 -38 -17z"/>
    <path d="M9600 10790 c0 -5 9 -10 20 -10 11 0 20 5 20 10 0 6 -9 10 -20 10
-11 0 -20 -4 -20 -10z"/>
    <path d="M9713 10785 c-27 -30 -10 -34 27 -5 23 18 23 20 5 19 -11 0 -26 -6
-32 -14z"/>
    <path d="M9795 10790 c-8 -13 5 -13 25 0 13 8 13 10 -2 10 -9 0 -20 -4 -23
-10z"/>
    <path d="M7488 6543 c6 -2 18 -2 25 0 6 3 1 5 -13 5 -14 0 -19 -2 -12 -5z"/>
    <path d="M9245 6538 c-24 -14 -71 -58 -61 -58 9 0 46 20 60 32 6 5 20 15 31
23 25 18 0 20 -30 3z"/>
  </g>
</svg>
```

De todos los objetos que existen (líneas, rectángulos, círculos, elipses, enlaces...) las imágenes que queremos analizar para este proyecto están totalmente formadas por recintos (o *paths*). Dentro de un elemento path se pueden definir trayectorias cerradas o abiertas a través de comandos, parámetros y coordenadas. En los recintos de nuestras imágenes nos encontraremos

básicamente tres comandos:

- **M ("moveto")**: Se desplaza hasta las coordenadas indicadas sin dibujar nada.
- **L ("lineto")**: Traza una línea recta desde el punto actual hasta las coordenadas indicadas.
- **C ("curveto")**: Dibuja una curva cúbica de Bézier desde el punto actual hasta el punto indicado, usando los otros dos puntos que recibe como puntos de control.

Los dos primeros son sencillos pero vamos a definir un poco más como se genera una curva de Bézier. Este tipo de curvas tienen una definición que las hace sencillas de utilizar en herramientas de dibujo, diseño CAD, diseño vectorial... Básicamente una curva de Bézier une dos puntos (puntos de anclaje o nodos), pero en lugar de hacerlo con una recta lo hace con una curva, y dicha curva tendrá una forma u otra basándose en los puntos invisibles que también la definen (puntos de control). Según el número de puntos de control que usemos podemos definir distintos órdenes de curvas de Bézier (lineales, cuadráticas, cúbicas...). Existe una expresión generalizada para cualquier curva de Bézier pero dado que nosotros solo trabajaremos con curvas cúbicas podemos quedarnos únicamente con la expresión de esta, ecuación 15.

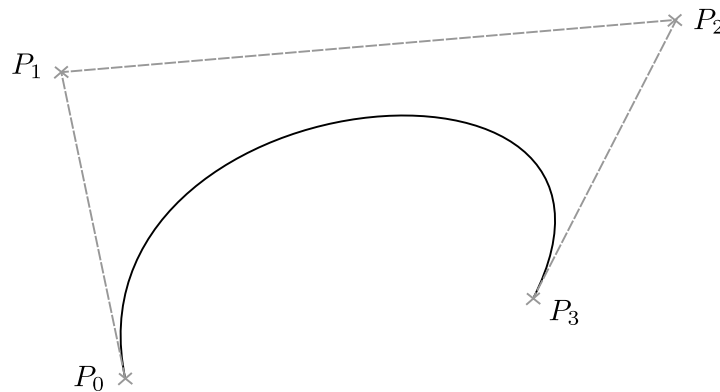


Figura 44: Curva cúbica de Bézier

$$\mathbf{B}(t) = \mathbf{P}_0(1-t)^3 + 3\mathbf{P}_1t(1-t)^2 + 3\mathbf{P}_2t^2(1-t) + \mathbf{P}_3t^3, \quad t \in [0, 1] \quad (15)$$

donde: $\mathbf{P}_0 = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix}$ es el punto inicial, $\mathbf{P}_3 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}$ es el punto final, $\mathbf{P}_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$ y $\mathbf{P}_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$ son los puntos de control y $\mathbf{B} = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$ es la curva cúbica de Bézier.

Aprovechándonos de lo expuesto se han creado las siguientes funciones de Python. El listado de código 4 utiliza la ecuación 15 para convertir la información obtenida sobre una curva cúbica de Bézier (puntos inicial, final y de control) junto con dos parámetros que definen la escala y la resolución en dos vectores con las coordenadas x e y de puntos que están dentro de la curva. Se podría decir que es una discretización de estas curvas. El efecto de esta función se esquematiza en la figura 45.

Listado de código 4 Función de Python para convertir una curva cúbica de Bézier en dos vectores de coordenadas discretas x e y

```
def bezier_to_points(p0, p1, p2, p3, t, res):
    # t: escala
    # res: resolución
```

```

bx = []
by = []
for i in np.arange(0, 1.001, res):
    x = np.real(p0)*(1-i)**3 + 3*i*np.real(p1)*(1-i)**2 + 3*np.real(p2)
    *(i**2)*(1-i) + np.real(p3)*(i**3)
    bx.append(t*x)
    y = np.imag(p0)*(1-i)**3 + 3*i*np.imag(p1)*(1-i)**2 + 3*np.imag(p2)
    *(i**2)*(1-i) + np.imag(p3)*(i**3)
    by.append(t*y)
return bx, by

```

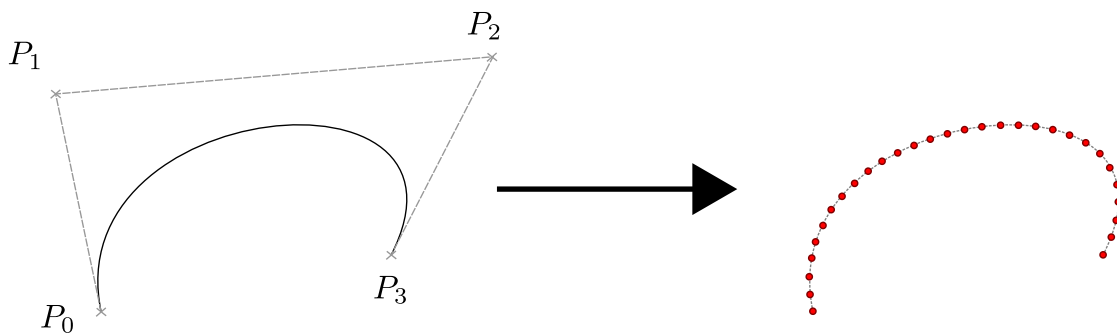


Figura 45: Discretización de una curva cúbica de Bézier

En los listados 5 y 6 se presentan dos funciones, la primera extrae los recorridos de cada path que encuentra dentro del archivo SVG que se está analizando y lo transforma en una concatenación de curvas de Bézier individuales definidas por cuatro puntos (inicial, final, y dos puntos de control), mientras que la segunda es capaz de analizar cada uno de estos recorridos y discretizar todas las curvas que encuentra con la función presentada anteriormente (listado 4) devolviendo dos vectores con las coordenadas x e y de todo el recorrido analizado.

Listado de código 5 Función de Python para analizar los recorridos de un archivo svg y extraer la información de las curvas que los conforman

```

from svg.path import parse_path
from lxml import etree
def extraer_recorridos(svg_directory):
    # svg_directory: ruta absoluta hasta el archivo svg
    svg_file = etree.parse(svg_directory)
    svg_root = svg_file.getroot()
    recorrido = []
    for index1 in range(len(svg_root)):
        if svg_root[index1].tag == '{http://www.w3.org/2000/svg}g':
            for index2 in range(len(svg_root[index1])):
                p = svg_root[index1][index2].attrib
                if 'd' in p:
                    recorrido.append(parse_path(svg_root[index1][index2].
attrib['d']))
    return recorrido

```

En la función *extraer_recorridos* nos aprovechamos de dos funciones, *parse_path* y *etree*, existentes en los módulos de Python *svg.path* y *lxml* respectivamente, los cuales importamos convenientemente antes de definir la función. Primero, con la función *etree* obtenemos el árbol con todas las etiquetas de un archivo tipo XML (como es el caso de los archivos SVG) y que nos permite escanear su información fácilmente. En segundo lugar, debemos reconocer las curvas que existen en una etiqueta concreta dentro de un archivo SVG. Como vemos la

función busca la etiqueta adecuada (la etiqueta $< g >$) en el archivo donde se encuentran definidas las curvas ($< path >$) y luego crea una lista llamada *recorrido* que va acumulando todas las curvas encontradas en orden.

Listado de código 6 Función de Python para discretizar todo un recorrido de un archivo svg

```
def puntos_contorno(path, esc, res):
    p_x = []
    p_y = []
    for curva in range(len(path)):
        if str(type(path[curva])) == "<class 'svg.path.path.CubicBezier'>":
            :
            a, b = bezier_to_points(path[curva].start, path[curva].
            control1, path[curva].control2, path[curva].end, esc,
            res)
            p_x.append(a)
            p_y.append(b)
            # if str(type(path[curva])) == "<class 'svg.path.path.Line'>":
            # ignorar (escribir accion si necesario)
    return p_x, p_y
```

Como se observa, en la función *puntos_contorno* se recorren todas las curvas de la lista proveniente de la función previa y se discretizan con la función definida anteriormente (*bezier_to_points*). Y es que en los archivos SVG generados con Potrace solo nos encontraremos con curvas de Bézier y líneas, sin embargo estas líneas no tienen dimensión (siempre van de un punto al mismo punto) y las genera la función *parse_path* para cerrar recorridos, por lo que no debemos extraer ningún punto de ellas.

5.3. Generar recorrido

Una vez podemos extraer la información de la imágenes vectoriales y discretizar los puntos de cada una de las curvas que las conforman vamos a tratar de diseñar el dibujo para que se ajuste a la superficie de trabajo. Debemos de poder definir a que altura se situará el plano de dibujo, el tamaño del dibujo y su colocación, así como la entrada al plano de dibujo, los cambios de trazo y la salida del actuador una vez se haya terminado el dibujo.

Para ello, en primer lugar se diseña una función, que aprovechando las funciones de extracción de información del apartado anterior, ajusta el dibujo a la posición y dimensión deseada en el plano. Además también devuelve un vector con la posición en la que se produce un cambio de trazo para poder añadir una trayectoria de desplazamiento en un plano superior al de dibujo. Esta función se muestra en el listado de código 7.

Listado de código 7 Función de Python para colocar y dimensionar los trazos del dibujo a realizar sobre el plano de trabajo

```
def dimensionado(r, ancho_deseado, alto_deseado, dibujar):
    # Transforma el conjunto de trayectorias de un dibujo en una sola
    # ajustado a un tamaño y centrada en el origen
    # r: recorridos a dimensionar
    # dibujar: si es 1 dibuja el resultado
    max_x = []
    min_x = []
    max_y = []
    min_y = []
    cambio_de_trazo = [] #recoge el indice del array donde se produce un
    cambio de path.
    indice = 0
    for i in range(len(r)):
```

```
px, py = puntos_contorno(r[i], 1, 0.05) #0.05
# plt.plot(px, py, 'ro')
# plt.show()
pmx = list(itertools.chain.from_iterable(px))
pmy = list(itertools.chain.from_iterable(py))
max_x.append(max(pmx))
min_x.append(min(pmx))
max_y.append(max(pmy))
min_y.append(min(pmy))
ancho = max(max_x) - min(min_x)
alto = max(max_y) - min(min_y)
if dibujar:
    print ancho, alto
escala_ancho = ancho_deseado / ancho
escala_alto = alto_deseado / alto
escala = min(escala_alto, escala_ancho)
if dibujar:
    print 'la escala es: %r' % escala
# SITUACION:
desp_x = (ancho * escala) / 2 + min(min_x) * escala
desp_y = (alto * escala) / 2 + min(min_y) * escala
# print "despx: %r despy: %r" % (desp_x, desp_y)
# DIBUJO ESCALADO Y COLOCADO EN EL CENTRO:
dib_x = []
dib_y = []
for i in range(len(r)):
    px, py = puntos_contorno(r[i], escala, 0.01) #0.01
    px = list(itertools.chain.from_iterable(px))
    py = list(itertools.chain.from_iterable(py))
    # Se guarda el indice donde se cambia de trazo para levantar el
    lapiz en el diseno de la trayectoria.
    dib_x.append(px)
    dib_y.append(py)
    indice += len(px)
    cambio_de_trazo.append(indice-1)
dib_x = list(itertools.chain.from_iterable(dib_x))
dib_y = list(itertools.chain.from_iterable(dib_y))
dib_x = [x - desp_x for x in dib_x]
dib_y = [y - desp_y for y in dib_y]
if dibujar:
    print "dibujo en x entre %r y %r siendo ancho %r" % (max(dib_x),
min(dib_x), max(dib_x) - min(dib_x))
    print "dibujo en y entre %r y %r siendo el alto %r" % (max(dib_y),
min(dib_y), max(dib_y) - min(dib_y))
    plt.plot(dib_x, dib_y, 'ro')
if dibujar:
    plt.show()
return dib_x, dib_y, cambio_de_trazo
```

Esta función nos devolverá las coordenadas de los puntos que conforman el dibujo sobre un plano plano XY sin definir. Ahora tenemos que definir una coordenada Z donde se realizará el dibujo y como realizar el acercamiento, cambio de trazo, y salida de este plano. En el caso del robot que ocupa este proyecto el plano de dibujo se ha colocado en $z = 215 \text{ mm}$, por lo tanto todas las coordenadas XY del dibujo generadas se completarán con esta coordenada Z. Algunas trayectorias generadas por esta función se muestran en la figura 46.

La siguiente función que se ha creado, es la función que aprovecha la información de donde se producen los cambios de trazo para añadir a la trayectoria los correspondientes ascensos y descensos para no dibujar en el recorrido del final de una trayectoria al inicio de la siguiente.

Esta función (listado de código 8). Recorre de forma inversa los vectores con las coor-

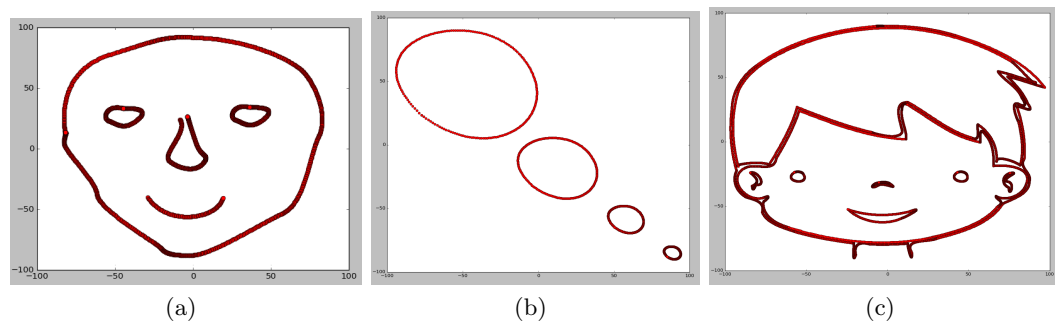


Figura 46: Trayectorias generadas en el plano de dibujo

denadas XYZ del dibujo para ir añadiendo una trayectoria de ascenso y descenso entre los puntos final e inicial de dos recorridos. Al recorrerse de forma inversa no se ven modificados los índices de los cambios de trazo previos y se reducen los cálculos y las variables a utilizar en esta tarea.

La trayectoria se realiza añadiendo puntos intermedios entre dos trazos en cuatro tramos:

- Un primer tramo de 10 puntos que avanza lentamente hacia al siguiente punto junto con una subida agresiva (25 *mm* por encima del plano de dibujo).
- Se continua avanzando más rápidamente y a una altura mayor (40 *mm* sobre el plano de trabajo hasta estar próximo al punto de destino).
- Se realiza una bajada donde se va decreciendo la coordenada Z linealmente según se finaliza la aproximación al punto destino.
- El último tramo simplemente añade 10 puntos extra con las coordenadas del destino. Esto se hace para dar un tiempo extra para corregir la posición en caso de que la bajada haya sido demasiado rápida (lo que ocurre cuando los puntos de origen y destino están considerablemente separados).

Listado de código 8 Función de Python para añadir a la trayectoria los cambios de trazo

```
def cambios_trazo(dx, dy, dz, cdt):
    # anyade puntos de subida en los cambios de trazo:
    for trazo in list(reversed(range(len(cdt) - 1))):
        # puntos iniciales y finales del cambio de trazo.
        x_ini = dx[cdt[trazo]]
        x_fin = dx[cdt[trazo] + 1]
        y_ini = dy[cdt[trazo]]
        y_fin = dy[cdt[trazo] + 1]
        # puntos intermedios:
        num_puntos = 200
        num_bajada = 20
        inc_x = (x_fin - x_ini) / num_puntos
        inc_y = (y_fin - y_ini) / num_puntos
        # insertar puntos:
        for j in range(num_puntos): # inserta puntos.
            dx.insert(cdt[trazo] + j + 1, x_ini + inc_x * j)
            dy.insert(cdt[trazo] + j + 1, y_ini + inc_y * j)
            if j < 10:
                dz.insert(cdt[trazo] + j + 1, pz - 25)
            elif j > (num_puntos - num_bajada):
```

```
        dz.insert(cdt[trazo] + j + 1, pz - (float(num_puntos - j)
/ num_bajada) * 40)
    else:
        dz.insert(cdt[trazo] + j + 1, pz - 40)
    for j in range(10):
        dx.insert(cdt[trazo] + num_puntos + j + 1, x_fin)
        dy.insert(cdt[trazo] + num_puntos + j + 1, y_fin)
        dz.insert(cdt[trazo] + num_puntos + j + 1, pz)
    return dx, dy, dz
```

Con toda la trayectoria de dibujo definida en coordenadas cartesianas XYZ debemos de traducir esta en posiciones angulares de los tres actuadores de nuestro robot. Para esta tarea nos servimos una vez más de la cinemática del robot (explicada en la sección 3). En el listado de código 9 se muestra una función sencilla que llama a la función de cinemática inversa definida anteriormente en cada punto de la trayectoria y almacena el resultado en tres vectores, uno para cada actuador.

Listado de código 9 Función de Python para convertir trayectorias cartesianas en trayectorias en posición angular de los actuadores

```
def conversion_angulos(dx, dy, dz):
    # convierte puntos cartesianos en angulos del motor.
    ang1 = []
    ang2 = []
    ang3 = []
    for i in range(len(dx)):
        angulos = dk.inverse(-dx[i], -dy[i], -dz[i])
        ang1.append(round(angulos[1], 1))
        ang2.append(round(angulos[3], 1))
        ang3.append(round(angulos[2], 1))
    return ang1, ang2, ang3
```

En este punto está diseñado todo el movimiento sobre el plano de trabajo (la trayectoria de dibujo), pero aún es necesario definir una aproximación inicial al punto inicial del dibujo en el plano de trabajo y una trayectoria de salida. De la primera tarea de encarga la función que se muestra en el listado de código 10. Esta función añade una trayectoria directamente definida en posiciones angulares de los tres actuadores, leen el punto inicial del dibujo y llegan hasta allí partiendo desde el punto origen con una trayectoria lineal.

Listado de código 10 Función de Python para realizar la aproximación al punto inicial del dibujo

```
def aproximacion_inicial(a1, a2, a3):
    # anyade aproximacion inicial para las trayectorias.
    longi = np.ceil(max(max(a1[0], a2[0]), a3[0])/0.1)
    l1 = list(np.arange(0, a1[0], a1[0]/longi))
    l1 = [round(elem, 2) for elem in l1]
    l2 = list(np.arange(0, a2[0], a2[0]/longi))
    l2 = [round(elem, 2) for elem in l2]
    l3 = list(np.arange(0, a3[0], a3[0]/longi))
    l3 = [round(elem, 2) for elem in l3]
    if len(l1) != len(l2) or len(l1) != len(l3) or len(l2) != len(l3):
        print 'vectores de distinto tamanho'
    if len(l1) != longi:
        l1.pop()
    if len(l2) != longi:
        l2.pop()
    if len(l3) != longi:
        l3.pop()
```

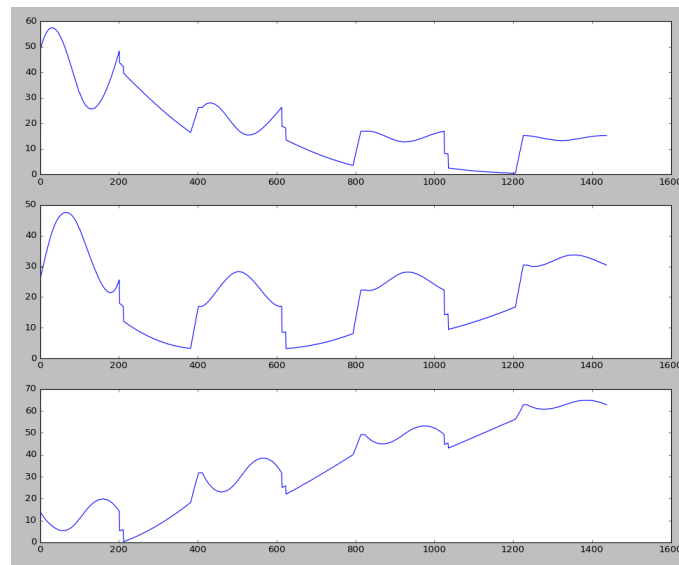


Figura 47: Trayectoria resultante para cada actuador. $\theta_1(t)$, $\theta_2(t)$ y $\theta_3(t)$ en orden descendente

```
sal1 = l1 + a1
sal2 = l2 + a2
sal3 = l3 + a3
return sal1, sal2, sal3
```

Mientras que de la segunda tarea para completar el cálculo del movimiento del robot se encarga la función que se presenta en el listado de código 11. Esta función genera una trayectoria de salida perpendicular al plano de trabajo desde el último punto de dibujo. Opcionalmente se le puede indicar que regrese a la posición origen del robot, $(\theta_1, \theta_2, \theta_3) = (0, 0, 0)$.

Listado de código 11 Función de Python para añadir a la trayectoria el levantamiento final y vuelta al origen

```
def salida_final(a1, a2, a3, tipo):
    #levanta el actuador al final de la trayectoria.
    p, p1, p2, p3 = dk.forward(a1[-1], a2[-1], a3[-1])
    pun_ini = [p1, p2, p3]
    inc = 40.0
    dim = 100
    p_x = [pun_ini[0] for indice in range(dim)]
    p_y = [pun_ini[1] for indice in range(dim)]
    p_z = [pun_ini[2] - indice * inc / dim for indice in range(dim)]
    s1, s2, s3 = conversion_angulos(p_x, p_y, p_z)
    sal1 = a1 + s1
    sal2 = a2 + s2
    sal3 = a3 + s3
    if tipo == 1:
        #levanta y despues vuelve al origen.
        o1, o2, o3 = mover(s1[-1], s2[-1], s3[-1], 0, 0, 0)
        sal1 += o1
        sal2 += o2
        sal3 += o3
    return sal1, sal2, sal3
```

De este modo ya se ha generado la trayectoria que seguirá el robot para realizar el dibujo como tres evoluciones temporales de las posiciones angulares de los actuadores, $\theta_1(t)$, $\theta_2(t)$ y $\theta_3(t)$.

y $\theta_3(t)$.

5.4. Conversión a trenes de pulsos

Una vez se ha definido una trayectoria completa, esta debe ser traducida en algo que Arduino entienda y que pueda generar el movimiento real de los tres motores que controlan los tres grados de libertad de nuestro motor delta.

En primer lugar no disponemos de una resolución infinita para fijar la posición de cada motor, como se ha comentado los motores paso a paso empleados se mueven en incrementos de $1,8^\circ$, pero, dado que utilizamos la técnica de *microstepping* reducimos esta paso hasta un octavo, siendo por lo tanto nuestro paso de $0,225^\circ$. De esta forma esto será lo que girará uno de nuestros motores paso a paso si le enviamos un pulso.

Y en segundo lugar debemos sincronizar el movimiento de todos los motores para que el movimiento a lo largo de la trayectoria sea lo más preciso posible. El objetivo ahora es convertir las trayectorias de posiciones angulares en trenes de pulsos sincronizados que muevan los motores. Serán necesarios 6 trenes de pulsos diferentes (dos por cada motor que deseamos controlar, para controlar ambas direcciones de giro).

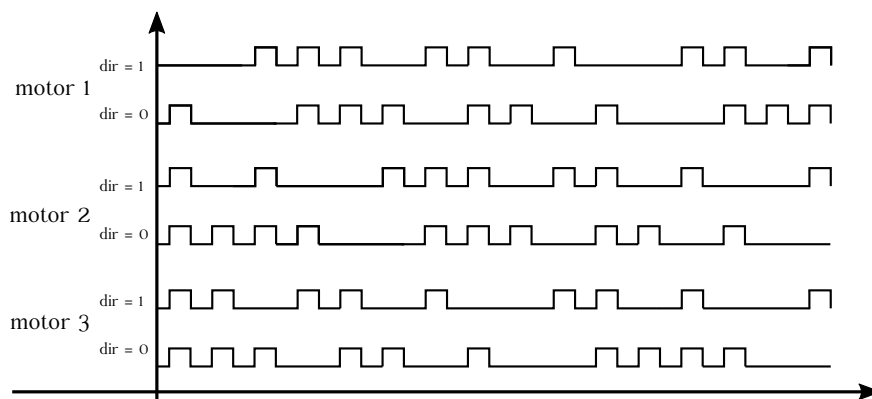


Figura 48: Representación de los trenes de pulsos sincronizados

La función que se encarga de esto es la que se presenta en el listado de código 12. Esta función va recorriendo toda la trayectoria generada con las anteriores funciones y generando pulsos para ir alcanzando cada punto. La condición para dar un punto por alcanzado y continuar con el siguiente es que todos los motores se encuentren a una distancia angular menor al doble del paso con el que trabajamos, puede parecer una condición laxa pero da resultados suficientemente precisos con unos tiempos de computación razonables.

Listado de código 12 Función de Python para generar los trenes de pulsos a partir de la trayectoria diseñada

```
def generar_pulsos(ang1, ang2, ang3, dibujar):  
    # convierte la trayectoria de posicion angular del motor en pulsos de  
    stepper  
    # return: pf (tren de pulsos forward), pb (tren de pulsos backwards)  
    step = 1.8/8  
    pf1 = [0]  
    pb1 = [0]  
    pf2 = [0]  
    pb2 = [0]  
    pf3 = [0]  
    pb3 = [0]  
    inc1 = 0.0
```

```
inc2 = 0.0
inc3 = 0.0
for i in range(1, len(ang1)):
    inc1 = inc1 + ang1[i] - ang1[i - 1]
    inc2 = inc2 + ang2[i] - ang2[i - 1]
    inc3 = inc3 + ang3[i] - ang3[i - 1]
    while True: # inc1 >= step or inc2 >= step or inc3 >= step:
        if inc1 >= step:
            pf1.append(1)
            pb1.append(0)
            inc1 -= step
        elif inc1 <= step:
            pf1.append(0)
            pb1.append(1)
            inc1 += step
        else:
            pf1.append(0)
            pb1.append(0)
    # 2
    if inc2 >= step:
        pf2.append(1)
        pb2.append(0)
        inc2 -= step
    elif inc2 <= step:
        pf2.append(0)
        pb2.append(1)
        inc2 += step
    else:
        pf2.append(0)
        pb2.append(0)
    # 3
    if inc3 >= step:
        pf3.append(1)
        pb3.append(0)
        inc3 -= step
    elif inc3 <= step:
        pf3.append(0)
        pb3.append(1)
        inc3 += step
    else:
        pf3.append(0)
        pb3.append(0)
    if inc1 < step*2 and inc2 < step*2 and inc3 < step*2:
        break
print 'incremento residual: %r' % inc1
print 'incremento residual: %r' % inc2
print 'incremento residual: %r' % inc3
# dibujar:
if dibujar != 0:
    abs = 0.0
    plt.subplot(3, 1, 1)
    # plt.plot(ang1)
    for k in range(len(pf1)):
        if pf1[k] == 1:
            abs += step
        if pb1[k] == 1:
            abs -= step
        plt.plot(k, abs, 'ro')
    abs = 0.0
    plt.subplot(3, 1, 2)
    # plt.plot(ang1)
    for k in range(len(pf2)):
```

```
    if pf2[k] == 1:
        abs += step
    if pb2[k] == 1:
        abs -= step
    plt.plot(k, abs, 'ro')
abs = 0.0
plt.subplot(3, 1, 3)
# plt.plot(ang1)
for k in range(len(pf3)):
    if pf3[k] == 1:
        abs += step
    if pb3[k] == 1:
        abs -= step
    plt.plot(k, abs, 'ro')
# plt.show()
return pf1, pb1, pf2, pb2, pf3, pb3
```

A modo de ejemplo se muestra en la figura 49 el movimiento que realizaría cada motor aplicando los pulsos generados a partir de las trayectorias de ejemplo de la figura 47 (incluye la aproximación inicial y la vuelta al origen del robot al final).

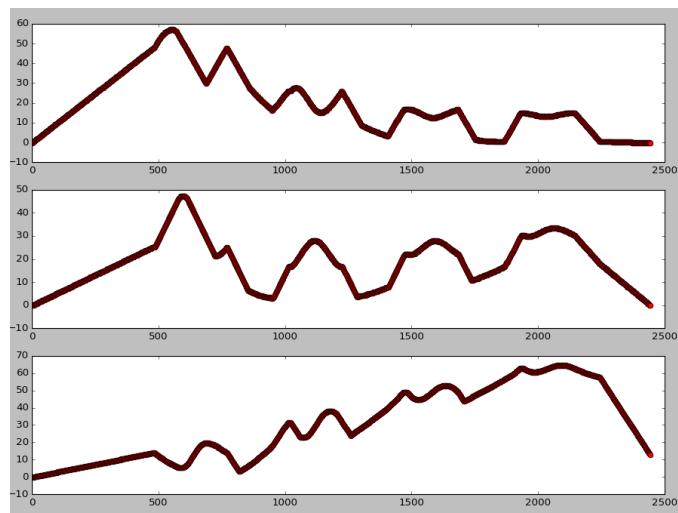


Figura 49: Movimiento generado en cada motor con trenes de pulsos

Tenemos listos ya los trenes de pulsos que debemos entregar a Arduino para seguir la trayectoria deseada, pero para ello es necesario comunicarse con Arduino de forma adecuada y pensar como enviar la información. Esta comunicación será realizada con ROS y será abordada en la siguiente sección.

6. Comunicación con ROS

Para la comunicación entre el ordenador, Arduino y el robot delta se empleará ROS, o *Robot Operating System*. ROS es un *framework* o infraestructura digital para escribir software robótico. Se trata de un conjunto de herramientas, librerías y convenciones que tienen como objetivo simplificar las tareas de creación de comportamientos complejos y robustos en robots a lo largo de una amplia variedad de plataformas robóticas.

ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. Está basado en una arquitectura de grafos donde el procesamiento toma lugar en nodos que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros.

La idea es generar los pulsos para los motores paso que producirán el movimiento controlado del robot tal y como se ha explicado en la sección anterior en un nodo, que se comunicará vía serial con otro nodo de Arduino que será el encargado de generar los pulsos a través del montaje ya explicado en la sección 4.

6.1. Paquete de control del robot

Empezamos creando un paquete nuevo que incluya el código del nodo (el cual generará la trayectoria y prepara los datos para ser enviados) y los nuevos mensajes necesarios.

Para enviar los pulsos al Arduino se decide enviar estos agrupándolos en bytes para hacer la comunicación más eficiente. Por esta razón se crea la función que se muestra en el listado de código 13. Esta función recibe como entrada un vector con todos los pulsos y los agrupa en bytes y convierte el grupo de ceros y unos en un entero positivo entre 0 y 255.

Listado de código 13 Función de Python para dividir los pulsos en grupos de 8

```
def division_pulsos(p):  
    #Agrupar de 8 en 8 los pulsos para enviarlos como integer.  
    p_int = []  
    for i in range(len(p) // 8):  
        cadena = ''  
        for j in range(i * 8, (i + 1) * 8):  
            cadena += str(p[j])  
        # print cadena  
        # print int(cadena, 2)  
        p_int.append(int(cadena, 2))  
    return p_int
```

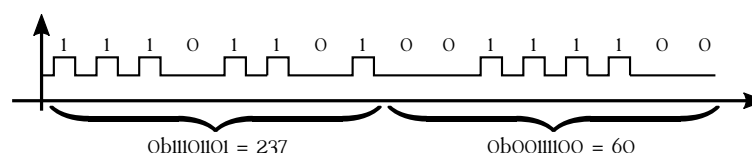


Figura 50: División en bytes

Nuestro nodo debe, una vez tiene toda la información lista para ser enviada, publicarla de modo correcto. Con este fin se define un nuevo mensaje de ROS con la forma que se muestra en el listado de código 14.

Listado de código 14 Nuevo mensaje de ROS Pulsos.msg

```
uint16 id
uint8 p1f
uint8 p1b
uint8 p2f
uint8 p2b
uint8 p3f
uint8 p3b
```

Los mensajes incluyen un "id" para identificar los pulsos y su posición en la trayectoria y evitar que se repitan o salten pulsos. Mientras que los 6 números siguientes se corresponden a un grupo de ocho periodos de pulsos de todos los trenes, el número que sigue a cada letra "p" indica el motor al que van dirigidos los pulsos y las letras "f" (*forward*) y "b" (*backwards*) indican la dirección de giro, bien hacia adelante (incrementando la posición angular) o bien hacia atrás (reduciendo la posición angular).

Una vez tenemos listo esto podemos escribir el script que generará el nodo encargado de generar la trayectoria y luego controlar el envío correcto de los bytes de pulsos a Arduino. Mostramos ahora el código que hace funcionar este nodo en el listado de código 15.

Listado de código 15 Script de Python que hace funcionar el nodo generador de trayectoria

```
#!/usr/bin/env python

import rospy
from std_msgs.msg import String
from tfm_delta.msg import Pulsos
from std_msgs.msg import UInt16
from tfm_delta.msg import trayectorias
from time import sleep
import generacion_trayectoria

pulso = Pulsos()

f1_int = []
f2_int = []
f3_int = []
b1_int = []
b2_int = []
b3_int = []

# Para recibir trayectorias en lugar de generarlas localmente:
trayec = trayectorias()
envio = trayectorias()

def nopulses():
    pulso.id = int(0)
    pulso.p1f = int(0)
    pulso.p1b = int(0)
    pulso.p2f = int(0)
    pulso.p2b = int(0)
    pulso.p3f = int(0)
    pulso.p3b = int(0)

# Para recibir trayectorias en lugar de generarlas localmente:
def callback(data):
    global trayec
    trayec = data
```

```
rospy.loginfo('recibido')

def publicar():
    # (...) Mostrada en otro listado de código.

if __name__ == '__main__':
    archivo = "/directorio/dela/imagen/para/trayectoria/imagen.svg"
    r = extraer_recorridos(archivo)
    pz = 215 # coordenada z del dibujo
    try:
        dx, dy, cdt = dimensionado(r, 190, 170, 1) #cdt: cambio de trazo
    200 180
        dz = [pz]*len(dx)

        # ANYADE PUNTOS DE SUBIDA EN LOS CAMBIOS DE TRAZO:
        dx, dy, dz = cambios_trazo(dx, dy, dz, cdt)
        dx, dy, dz = entrada_inicial(dx, dy, dz)

        # CONVERTIR PUNTOS EN ANGULOS DEL MOTOR
        ang1, ang2, ang3 = conversion_angulos(dx, dy, dz)

        # APROXIMACION AL PUNTO INICIAL:
        ang1, ang2, ang3 = aproximacion_inicial(ang1, ang2, ang3)

        # SALIDA FINAL:
        ang1, ang2, ang3 = salida_final(ang1, ang2, ang3, 1)

        # PRODUCIR TRENES DE PULSOS PARA CADA MOTOR
        f1, b1, f2, b2, f3, b3 = generar_pulsos(ang1, ang2, ang3, 1)

        # ENVIAR TRENES DE PULSOS:

        # agrupar por bytes
        l = np.ceil(len(f1)/8.0)*8
        for i in range(int(l-len(ang1))):
            f1.append(0)
            f2.append(0)
            f3.append(0)
            b1.append(0)
            b2.append(0)
            b3.append(0)

        f1_int = division_pulsos(f1)
        f2_int = division_pulsos(f2)
        f3_int = division_pulsos(f3)
        b1_int = division_pulsos(b1)
        b2_int = division_pulsos(b2)
        b3_int = division_pulsos(b3)

        publicar()

    except rospy.ROSInterruptException:
        pass
```

A continuación vamos a ir comentando las distintas partes de código que son necesarias para hacer funcionar este nodo:

- **Importar los módulos y paquetes necesarios:** En las primeras líneas de código se añaden módulos y paquetes necesarios para hacer funcionar el nodo.

- *rospy*: es una librería que contiene una serie de funciones y objetos que facilitan y simplifican la programación de nodos, publicadores y subscriptores entre otros elementos de ROS.
 - paquetes *.msg*: Permiten el uso de mensajes típicos de ROS así como de los propios creados por el usuario.
 - la función *sleep*: esta función suspende la ejecución del programa por un número determinado de segundos.
 - *generacion_trayectoria*: módulo que contiene todas las funciones para la generación de trayectoria descritas en la sección 5.
- **Creación de variables:** Se inicializan variables que serán usadas de forma global y se generan objetos con la forma de los mensajes de ROS que vamos a utilizar. Adicionalmente también se generan variables para recibir trayectorias (con el formato adecuado) en lugar de generarlas localmente.
- **Funciones adicionales:** Se añaden tres funciones más.
- *nopulses()*: Función que limpia y pone a cero todos los trenes de pulsos enviados. Se crea con el objetivo de reducir las líneas de código que supone hacer una acción tan sencilla en la función principal.
 - *callback()*: Función opcional, necesaria si se desea recibir la trayectoria en lugar de generarla localmente. Esta función será llamada cada vez que exista un mensaje nuevo en el tema de ROS al que se subscriba el nodo.
 - *publicar()*: Función que se encargue de manejar las comunicaciones (crear el nodo, iniciar un *publisher* y controlar la frecuencia de envío). Se muestra en el listado de código 16 y se comentará en detalle a continuación.
- **Función principal:** Función que contiene las instrucciones principales del nodo.
- En primer lugar carga el archivo de la imagen SVG a partir de la cual se generará la trayectoria y se define el plano de dibujo (en este caso $z = 215 \text{ mm}$).
 - Se llaman a las funciones explicadas en la sección 5 definiendo el tamaño del dibujo en la función *dimensionado()*.
 - Se agrupan los trenes de pulsos en paquetes de un byte.
 - Se llama a la función que conduce el el nodo y la comunicación con ROS.

Vamos a detenernos ahora a analizar la función que gobierna el nodo (Listado de código 16).

Listado de código 16 Función que se encarga de generar y controlar el nodo.

```
def publicar():
    rospy.init_node('generador_pulsos', anonymous=False)
    pub = rospy.Publisher('trenes_pulsos', Pulsos, queue_size=10)
    pub2 = rospy.Publisher('num_pulso', UInt16, queue_size=10)
    rate = rospy.Rate(7.8125)
    # Para recibir trayectorias en lugar de generarlas localmente:
    #rospy.Subscriber("trayectorias", trayectorias, callback)
    i = 0
    sleep(5)
    while not rospy.is_shutdown():
        pulso.id = i
        pulso.p1f = f1_int[i-1]
```

```
pulso.p1b = b1_int[i-1]
pulso.p2f = f2_int[i-1]
pulso.p2b = b2_int[i-1]
pulso.p3f = f3_int[i-1]
pulso.p3b = b3_int[i-1]
if i < len(f1_int):
    rospy.loginfo(i)
    i += 1
pub.publish(pulso)
if i >= len(f1_int):
    nopulses()
    rospy.loginfo('trayectoria finalizada')
pub2.publish(i)
rate.sleep()
```

Distinguimos las siguientes acciones en esta función:

- En primer lugar se inicia el nodo indicando su nombre, en este caso *generador_pulsos*, y que no hay necesidad de generar un nombre único para este (*anonymous=False*).
- Después se inicializa un publicador (*publisher*) que se encargará de publicar un tipo de mensajes especificado (*Pulsos*) en un cierto tema (*trenes_pulsos*). Especificamos además el tamaño máximo de mensajes que se pueden almacenar en cola antes de ser enviados.
- Comentado aparece la forma de crear un subscriptor al tema *trayectorias*, para recibir las trayectorias a través de este tema en lugar de generarlas localmente.
- Se define la frecuencia del bucle que se corresponde con la velocidad a la que se enviarán los bytes de pulsos. En Arduino el periodo de funcionamiento es de $T_{\text{Arduino}} = 16 \text{ ms}$, que resultó ser el periodo más rápido al que se funcionaba con una robustez suficiente. Conociendo esto podemos realizar un cálculo sencillo (ecuación 16) para conocer la frecuencia a la que se deben enviar los bytes de pulsos y que el conjunto funcione bien sincronizado.

$$f_{\text{envio}} = \frac{1}{T_{\text{envio}}} = \frac{1}{T_{\text{Arduino}} \left(\frac{8 \text{ pulsos}}{\text{envio}} \right)} = \frac{1}{16 \text{ ms} \cdot 8} = 7,8125 \text{ Hz} \quad (16)$$

- Antes de comenzar el envío de pulsos se realiza una pequeña pausa de 5 s para que haya un pequeño margen para quitar las fijaciones del robot si aún no habían sido retiradas.
- Finalmente se inicia el bucle principal del nodo. Este se encarga de rellenar el mensaje a enviar con los bytes de pulsos y un id correspondiente a la posición en las listas que se está enviando y además se imprime un mensaje en la terminal con esta posición. Cuando se ha recorrido todas las listas se imprime en la terminal un mensaje que indica que se ha terminado la trayectoria. La última instrucción *rate.sleep()* suspende la ejecución del programa el tiempo necesario para que se consiga que el bucle tenga el periodo deseado y se mantenga la frecuencia definida previamente.

En este punto se están enviando y publicando correctamente los trenes de pulsos en un tema de ROS. En próximo paso será que se esta información que está disponible sea aprovechada por otro nodo para transformarla en movimiento real del robot.

6.2. Paquete roserial y Arduino

Una vez se envían los pulsos de forma correcta tenemos que conseguir que nuestra placa Arduino UNO sea capaz de recibir la información y ejecutar el movimiento de los motores.

Con este fin se hace uso de un paquete ya existente en ROS y que facilita las comunicaciones a través del puerto serie, se trata de *rosserial*.

El paquete *rosserial* y sus derivados *rosserial_arduino* y *rosserial_python* incluyen las herramientas y scripts necesarios para gobernar las comunicaciones a través del puerto serie. En el último de ellos encontramos un script denominado "*serial_node.py*" que inicia las comunicaciones con el dispositivo conectado a través del puerto serie, iniciando automáticamente los nodos, subscriptores y publicadores definidos conforme a la información almacenada en el dispositivo al que se está conectando, en nuestro caso una placa Arduino UNO.

Por otro lado en el paquete *rosserial_arduino* encontramos la funcionalidad necesaria para generar los encabezamientos (en inglés *headers*) necesarios para que Arduino y las librerías de ROS sean capaces de interpretar los nuevos mensajes personalizados que hemos generado. El script empleado con este fin se llama "*make_libraries.py*".

En el listado de código 17 se muestra el *sketch* de Arduino que funcionará como nodo ejecutor y que enviará los pulsos a los motores para que estos se muevan conforme a la trayectoria diseñado en el nodo descrito anteriormente.

Listado de código 17 Función Arduino que recibe los trenes de pulsos y los ejecuta

```
//ROS:
#include <ros.h>
#include <tfm_delta/Pulsos.h>
#include <std_msgs/UInt8.h>
5  #include <tfm_delta/stepper_pos.h>
#include <TimerOne.h>

//Pines drivers:
int en1 = 2;
10 int pul1 = 3;
int dir1 = 4;
int en2 = 5;
int dir2 = 6;
int pul2 = 7;
15 int en3 = 8;
int pul3 = 9;
int dir3 = 10;

//ros:
20 ros::NodeHandle_<ArduinoHardware, 2, 1, 280, 280> nh;

tfm_delta::Pulsos recibido;
tfm_delta::stepper_pos posicion;

25 boolean nuevo = 0;
boolean done = 0;

void messageCb( const tfm_delta::Pulsos& escuchado){
    if (recibido.id != escuchado.id){
30         recibido = escuchado;
        nuevo= 1;
    }
}

35 ros::Subscriber<tfm_delta::Pulsos> sub("trenes_pulsos", &messageCb );
//ros::Publisher pub("pulsos_arduino", &recibido);
ros::Publisher posicion_stepper("pos_steppers", &posicion);

//driver:
40 boolean control_loop = 0;
int i = 0;
```

```
byte p1f = 0;
byte p1b = 0;
byte p2f = 0;
45 byte p2b = 0;
byte p3f = 0;
byte p3b = 0;

int ancho_pulso = 500;
50 int post_pulso = 500;

void adelante1(){
    digitalWrite(4,1);
    digitalWrite(3,!digitalRead(3));
55    delayMicroseconds(ancho_pulso);
    digitalWrite(3,!digitalRead(3));
    delayMicroseconds(post_pulso);
}

60 void atras1(){
    digitalWrite(dir1,0);
    digitalWrite(3,!digitalRead(3));
    delayMicroseconds(ancho_pulso);
    digitalWrite(3,!digitalRead(3));
65    delayMicroseconds(post_pulso);
}

void adelante2(){
    digitalWrite(en2,1);
70    digitalWrite(dir2,1);
    digitalWrite(pul2,!digitalRead(pul2));
    delayMicroseconds(ancho_pulso);
    digitalWrite(pul2,!digitalRead(pul2));
    delayMicroseconds(post_pulso);
75 }

void atras2(){
    digitalWrite(en2,1);
    digitalWrite(dir2,0);
80    digitalWrite(pul2,!digitalRead(pul2));
    delayMicroseconds(ancho_pulso);
    digitalWrite(pul2,!digitalRead(pul2));
    delayMicroseconds(post_pulso);
}

85 void adelante3(){
    digitalWrite(dir3,1);
    digitalWrite(pul3,!digitalRead(pul3));
    delayMicroseconds(ancho_pulso);
90    digitalWrite(pul3,!digitalRead(pul3));
    delayMicroseconds(post_pulso);
}

void atras3(){
95    digitalWrite(dir3,0);
    digitalWrite(pul3,!digitalRead(pul3));
    delayMicroseconds(ancho_pulso);
    digitalWrite(pul3,!digitalRead(pul3));
    delayMicroseconds(post_pulso);
100 }

void origen() {
    adelante1();
```

```
    adelante2();
105    adelante3();
}

void setup()
{
110    //ros:
    nh.getHardware()->setBaud(115200);
    nh.initNode();
    nh.subscribe(sub);
    // nh.advertise(pub);
115    nh.advertise(posicion_stepper);
    //driver:
    pinMode(en1,OUTPUT);
    pinMode(pul1,OUTPUT);
    pinMode(dir1,OUTPUT);
120    pinMode(en2,OUTPUT);
    pinMode(pul2,OUTPUT);
    pinMode(dir2,OUTPUT);
    pinMode(en3,OUTPUT);
    pinMode(pul3,OUTPUT);
125    pinMode(dir3,OUTPUT);
    origen();
    posicion_stepper1 = 0;
    posicion_stepper2 = 0;
    posicion_stepper3 = 0;
130    //timer:
    Timer1.initialize(16000); //16ms
    Timer1.attachInterrupt(ISR_Blink);
}

135 void ISR_Blink(){
    if (control_loop == 0){
        control_loop = 1;
    }
    else {
140        // error:
        nh.logwarn("error, se ha excedido el tiempo de ciclo.");
    }
}

145 void loop()
{
    if (control_loop == 1) {

150        if(nuevo==1){
            p1f = recibido.p1f;
            p1b = recibido.p1b;
            p2f = recibido.p2f;
            p2b = recibido.p2b;
            p3f = recibido.p3f;
155            p3b = recibido.p3b;
            nuevo = 0;
            done = 0;
        }

160        if(done == 0){
            if(p1f>>i & 0b1){
                adelante1();
                posicion_stepper1++;
165            }
        }
    }
}
```

```
        if(p1b>>i & 0b1){
            atras1();
            posicion.stepper1--;
        }
170
        if(p2f>>i & 0b1){
            adelante2();
            posicion.stepper2++;
        }
175
        if(p2b>>i & 0b1){
            atras2();
            posicion.stepper2--;
        }

180
        if(p3f>>i & 0b1){
            adelante3();
            posicion.stepper3++;
        }
        if(p3b>>i & 0b1){
185
            atras3();
            posicion.stepper3--;
        }

        i++;
190
        if(i>7){
            i=0;
            done = 1;
        }
    }
195

    if(done == 1){
        nh.spinOnce();
        posicion_stepper.publish( &posicion);
    }
200

    control_loop = 0;
}
}
```

Vamos a ir comentar el código con detenimiento:

- Líneas 1-6: Se incluye la librería de ROS, los encabezamientos de los mensajes que vamos a utilizar (*Pulsos.h* y *stepper_pos.h* generados anteriormente) y la librería *TimerOne* que simplifica el uso de los *timers* de nuestra placa Arduino.
- Líneas 9-17: Pines a los que se conectan las señales de control (pulso, dirección y habilitación) de las controladoras que contienen los circuitos de potencia para mover los motores.
- Línea 20: *ros::NodeHandle* es una clase que permite iniciar un nodo en ROS fácilmente dentro de un programa de C++, además facilita (a través de una nueva capa en forma de *namespace* que facilita la escritura de subcomponentes. *<ArduinoHardware, 2, 1, 280, 280>* nos permite especificar el número de publicadores, el número de subscriptores, y el tamaño de los buffers de cada uno de ellos respectivamente en bytes.
- Líneas 22 y 23: Inicializa variables con la forma de los mensajes de ROS.
- Líneas 25 y 26: Dos variables binarias para el control del flujo de programa.

- Líneas 28-33: Función que se llama cada vez que se publica un mensaje en el tema al que nos subscribimos. Esta función actualiza los pulsos recibidos y indica que son nuevos pulsos si su identificador es diferente.
- Líneas 35-37: Se inicializan los objetos de los subscriptores y publicadores, indicando en su declaración el tipo de mensaje que utilizan, el nombre del objeto, el nombre del tema de ROS donde se publica o lee información y por último una función de "callback" en el caso de los subscriptores y la variable a publicar en el caso de los publicadores.
- Líneas 40-47: Declaración de distintas variables de control y para almacenar los pulsos.
- Líneas 49 y 50: Estas variables fijan el tamaño de los pulsos enviados especificando el tiempo activo y otro tiempo post-pulso en que obligatoriamente no habrá pulsos. Los tiempos están especificados en milisegundos.
- Líneas 52-100: Estas seis funciones ejecutan todos los pulsos posibles que se pueden enviar, dos para cada motor (uno en la dirección creciente de la posición angular y otro decreciente). Los pulsos generados son de forma cuadrada, utilizando las variables del punto anterior, con un tiempo activo y otro tiempo inactivos. En vez de indicar si el tiempo activo es con una señal digital alta (*HIGH*) o baja (*LOW*) se niega la señal actual, de esta forma todas las funciones tienen el mismo patrón indiferentemente de la controladora de cada motor, ya que unas utilizan el valor 0 V como referencia y otras los 5 V.
- Líneas 102-106: La función `origen()` cumple una función muy sencilla y es enviar un pulso a cada motor cuando es llamada. Al usarla en el bloque *setup* hace que se bloqueen los motores en su posición inicial al cargar el programa.
- Bloque *setup*:
 - Líneas 111-115: En primer lugar se indica al objeto *ros::NodeHandle nh* definido al principio la velocidad a la que va a funcionar el puerto serie, por la corta distancia a la que transmitimos podemos emplear la velocidad más alta para Arduino (115200 baud). Después inicializamos el nodo de ROS, los subscriptores y publicadores que vayamos a utilizar.
 - Líneas 117-125: Se configura todos los pines conectados a las controladoras de los motores como pines de salida.
 - Líneas 126-129: Se llama a la función `origen` que bloquea los motores en el inicio y se establece la posición actual como (0, 0, 0).
 - Líneas 131 y 132: Configuramos una interrupción asociada a un *timer* que se utilizará para forzar el periodo del bucle principal. El período se establece en $T = 16\text{ ms}$, que resultó ser, tras aplicar un método heurístico, el periodo más rápido al que podía funcionar con robustez el nodo.
- Líneas 135-143: Función que se llama cuando se produce la interrupción del *Timer1* definido anteriormente. Permite reiniciar el bucle principal si este ha concluido todas sus acciones, en caso contrario muestra un aviso en la terminal donde esta corriendo indicando que no se está respetando la restricción de tiempo para el bucle.
- Bloque *loop*:
 - Líneas 147 y 202: Instrucciones que complementan lo explicado anteriormente, y que fuerzan las instrucciones dentro de este bloque *if* a ejecutarse cada 16 ms.

- Líneas 150-159: Actualiza los bytes de pulsos si se trata de un envío nuevo y después reinicia dos variables para controlar el flujo del programa.
- Líneas 161-189: Se encarga de ejecutar todos los pulsos del último envío. Por orden va recorriendo cada bit (aprovechando de operaciones de desplazamiento en los bytes recibidos) del último envío y enviando el pulso correspondiente cuando un bit es un 1 y no realiza nada si es un 0.
- Líneas 190-194: Si la variable i es mayor que 7 quiere decir que ya se han recorrido todos los bits de los bytes actuales por lo que se resetea la variable y se indica a la variable *done* de control de flujo que se ha terminado.
- Líneas 196-199: Se escucha al tema *trenes_pulsos* y mientras tanto también se publica la posición actual de los tres motores.
- Línea 201: Se resetea la variable de control que se encarga de asegurar que el período sea el estipulado.

6.3. Paquete de visualización

Para confirmar el correcto funcionamiento y robustez de la trayectoria y pulsos generados se desarrolla un paquete de visualización en ROS, con el que visualizar las trayectorias sin necesidad de mover el motor real y evitar fallos en este que provoquen roturas por movimientos inesperados fuera de los límites de trabajo del robot.

Con este fin se va a utilizar *RVIZ* que es la herramienta de visualización que incorpora ROS. Se debe en primer lugar crear un modelo virtual del robot que sea soportado por esta herramienta, el formato típicamente usado en ROS con este fin es el formato *URDF* (*The Universal Robotic Description Format*). Se trata de un formato de archivos tipo XML usado para describir todos los elementos de un robot (generalmente con elementos geométricos sencillos) a través de articulaciones (*joints*) y enlaces (*links*) relativamente en una estructura de árbol.

Los enlaces no son más que ejes de coordenadas relativos para una parte de un robot, representada por elementos geométricos sencillos asociados a cada enlace (en la etiqueta *visual*) y que se sitúan con relación a los ejes de coordenadas de su enlace. Las articulaciones se sitúan en relación a un enlace "padre" y marcan la posición del eje de coordenadas de un enlace "hijo" así como el tipo de movimiento que puede realizar la articulación, existiendo articulaciones de revolución, continuas, prismáticas, fijas, flotantes y planas.

Una de las restricciones que nos encontramos con este formato es que un enlace puede ser "padre" de varios enlaces pero solo puede ser "hijo" de un único enlace, en consecuencia, encadenando articulaciones con esta estructura se pueden crear cadenas cinemáticas abiertas en forma de árbol pero nunca podremos cerrar la estructura de articulaciones. El robot delta que ocupa este proyecto es una estructura cerrada, así que para poder realizar una visualización del mismo debemos de definirlo de una forma menos realista en cuanto a la verdadera estructura de sus articulaciones pero que nos permita reproducir su movimiento de forma correcta. La estructura tomada es la siguiente:

- La **base superior** del robot será el enlace base que se situará en el origen absoluto de coordenadas. Su representación es cilindro sencillo de color gris. Contará con cuatro enlaces hijo:
 - Los **tres brazos superiores** se representan a través de prismas de dimensiones similares a las reales a las de los brazos reales, su colocación se obtiene a través del script de Matlab diseñado en la sección 3 del que obtenemos todos los puntos geométricos significativos, y las articulaciones que los unen a la base superior son articulaciones de revolución en el eje Y.

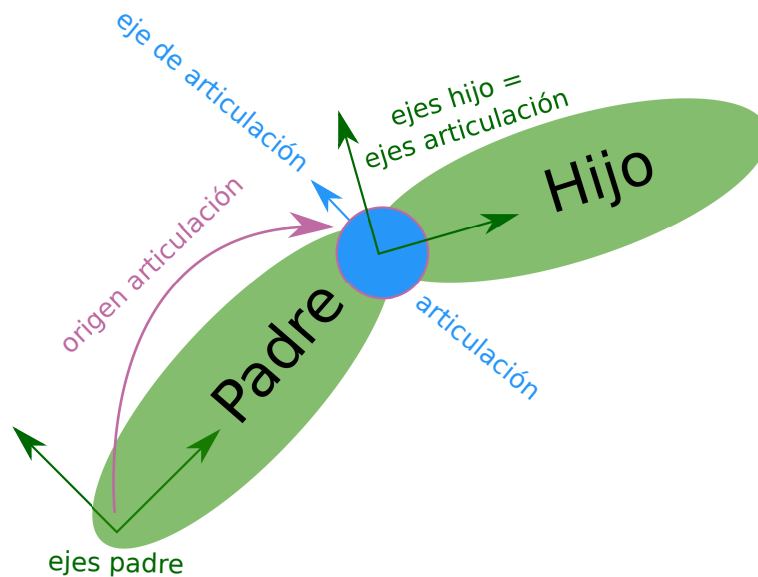


Figura 51: Esquema de enlaces padre e hijo unidos por articulación en formato URDF

- **Tres enlaces vacíos** (sin elemento visual) se utilizan para colocar apropiadamente los brazos inferiores. Se unen a los brazos superiores al final de estos y a través de una articulación de revolución en el eje Y. Una vez más la información geométrica concreta propia de nuestro robot delta se obtiene a través de Matlab y se introduce en el archivo *URFD*.
 - ◊ El final de estas tres cadenas que conforman los tres brazos del robot son los enlaces que representan los **tres brazos inferiores** unidos a los anteriores mediante articulaciones de revolución en el eje Z. La representación visual son sencillos prismas de dimensiones similares al brazo real.
- Un **primer enlace auxiliar** se utilizará para colocar el actuador en su posición (esto forma parte de las modificaciones respecto de la estructura real para poder realizar la visualización). No tiene ningún elemento visual asociado y su articulación es de tipo prismática y se encarga del movimiento en la dirección X.
 - **Segundo enlace auxiliar** unido al primero e igual que este sin elemento visual. Es una articulación prismática para controlar el movimiento en la dirección Y.
 - ◊ El **Actuador** aparece al final de toda esta cadena auxiliar un con una representación sencilla en forma un cilindro azul asociada a el, el cual termina de colocar con la articulación prismática con movimiento en la dirección Z que lo controla.

En el listado de código 18 vemos como se ha definido el archivo URDF.

Listado de código 18 Representación en formato URDF del robot delta

```
<?xml version="1.0"?>
<robot name="Delta_robot">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.05" radius="0.04"/>
      </geometry>
      <material name="gris">
```



```
<color rgba="0.55 0.55 0.55 1" />
</material>
</visual>
</link>

<!--BRAZOS SUPERIORES:-->
<link name="brazo1">
  <visual>
    <origin xyz="0.075 0 0" rpy="0 0 0"/> <!--la = 150-->
    <geometry>
      <box size="0.15 0.03 0.03"/> <!--la = 150-->
    </geometry>
    <material name="blanco">
      <color rgba="0.9 0.9 0.9 1" />
    </material>
  </visual>
</link>
<joint name="base_brazo1" type="revolute">
  <axis xyz="0 1 0"/>
  <limit effort="100.0" lower="0.0" upper="1.8" velocity="0.5" />
  <parent link="base.link" />
  <child link="brazo1" />
  <origin xyz="0.0360844 0 0"/> <!--A1 = 36.0844 0 0-->
</joint>
<link name="brazo2">
  <visual>
    <origin xyz="0.075 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.15 0.03 0.03"/> <!--la = 150-->
    </geometry>
    <material name="blanco" />
  </visual>
</link>
<joint name="base_brazo2" type="revolute">
  <axis xyz="0 1 0"/>
  <limit effort="100.0" lower="0.0" upper="1.8" velocity="0.5" />
  <parent link="base.link" />
  <child link="brazo2" />
  <origin xyz="-0.0180422 -0.03125 0" rpy="0 0 -2.0944"/> <!--A2 =old -31.0326 -53.7500
0-->
</joint>
<link name="brazo3">
  <visual>
    <origin xyz="0.075 0 0" rpy="0 0 0"/>
    <geometry>
      <box size="0.15 0.03 0.03"/> <!--la = 150-->
    </geometry>
    <material name="blanco" />
  </visual>
</link>
<joint name="base_brazo3" type="revolute">
  <axis xyz="0 1 0"/>
  <limit effort="100.0" lower="0.0" upper="1.8" velocity="0.5" />
  <parent link="base.link" />
  <child link="brazo3" />
  <origin xyz="-0.0180422 0.03125 0" rpy="0 0 2.0944"/> <!--A3 = -31.0326 53.7500 0-->
</joint>

<!--BRAZOS INFERIORES:-->
<link name="barra1_a" />
<joint name="codo1_a" type="revolute">
```

```
<axis xyz="0 -1 0" />
<limit effort="100.0" lower="-3.14" upper="3.14" velocity="0.5" />
<parent link="brazo1" />
<child link="barra1_a" />
<origin xyz="0.15 0 0" rpy="0 0 0" /> <!--02 = 62.0652 -107.5000 0-->
</joint>
<link name="barra1_b">
  <visual>
    <origin xyz="-0.1025 0 0" rpy="0 0 0" /> <!--0.175-->
    <geometry>
      <box size="0.205 0.02 0.02" /> <!--1b = 205-->
    </geometry>
    <material name="blanco" />
  </visual>
</link>
<joint name="codo1_b" type="revolute">
  <axis xyz="0 0 1" />
  <limit effort="100.0" lower="-3.14" upper="3.14" velocity="0.5" />
  <parent link="barra1_a" />
  <child link="barra1_b" />
  <origin xyz="0 0 0" rpy="0 0 0" /> <!--02 = 62.0652 -107.5000 0-->
</joint>

<link name="barra2_a" />
<joint name="codo2_a" type="revolute">
  <axis xyz="0 -1 0" />
  <limit effort="100.0" lower="-3.14" upper="3.14" velocity="0.5" />
  <parent link="brazo2" />
  <child link="barra2_a" />
  <origin xyz="0.15 0 0" rpy="0 0 0" /> <!--02 = 62.0652 -107.5000 0-->
</joint>
<link name="barra2_b">
  <visual>
    <origin xyz="-0.1025 0 0" rpy="0 0 0" />
    <geometry>
      <box size="0.205 0.02 0.02" /> <!--1b = 205-->
    </geometry>
    <material name="blanco" />
  </visual>
</link>
<joint name="codo2_b" type="revolute">
  <axis xyz="0 0 1" />
  <limit effort="100.0" lower="-3.14" upper="3.14" velocity="0.5" />
  <parent link="barra2_a" />
  <child link="barra2_b" />
  <origin xyz="0 0 0" rpy="0 0 0" /> <!--02 = 62.0652 -107.5000 0-->
</joint>

<link name="barra3_a" />
<joint name="codo3_a" type="revolute">
  <axis xyz="0 -1 0" />
  <limit effort="100.0" lower="-3.14" upper="3.14" velocity="0.5" />
  <parent link="brazo3" />
  <child link="barra3_a" />
  <origin xyz="0.15 0 0" rpy="0 0 0" /> <!--02 = 62.0652 -107.5000 0-->
</joint>
<link name="barra3_b">
  <visual>
    <origin xyz="-0.1025 0 0" rpy="0 0 0" />
    <geometry>
      <box size="0.205 0.02 0.02" /> <!--1b = 205-->
    </geometry>
```

```

    <material name="blanco" />
  </visual>
</link>
<joint name="codo3_b" type="revolute">
  <axis xyz="0 0 1" />
  <limit effort="100.0" lower="-3.14" upper="3.14" velocity="0.5" />
  <parent link="barra3_a" />
  <child link="barra3_b" />
  <origin xyz="0 0 0" rpy="0 0 0" /> <!--02 = 62.0652 -107.5000 0-->
</joint>

<!--ACTUADOR:-->
<link name="actuador">
  <visual>
    <origin xyz="0 0 0" rpy="0 -1.57075 0" />
    <geometry>
      <cylinder length="0.01" radius="0.02" />
    </geometry>
    <material name="colorte">
      <color rgba="0.15 0.55 0.95 1" />
    </material>
  </visual>
</link>
<link name="aux1" />
<link name="aux2" />
<joint name="act_x" type="prismatic">
  <parent link="base.link" />
  <child link="aux1" />
  <limit effort="100.0" lower="-1" upper="1" velocity="0.5" />
  <origin rpy="0 0 0" xyz="0 0 0" />
</joint>
<joint name="act_y" type="prismatic">
  <parent link="aux1" />
  <child link="aux2" />
  <limit effort="100.0" lower="-1" upper="1" velocity="0.5" />
  <origin rpy="0 0 -1.57075" xyz="0 0 0" />
</joint>
<joint name="act_z" type="prismatic">
  <parent link="aux2" />
  <child link="actuador" />
  <limit effort="100.0" lower="0" upper="1" velocity="0.5" />
  <origin rpy="0 1.57075 0" xyz="0 0 0" />
</joint>

</robot>

```

Para lanzar el modelo en RVIZ es necesario realizar un pequeño código *launch* indicando algunos parámetros, argumentos, dando nombre al nodos y llamando a un archivo RVIZ que contenga la información de la cámara y entorno donde se visualizará el robot. El archivo *launch* se muestra en el listado de código 19 y en la figura 52 se puede ver el resultado.

Listado de código 19 Archivo launch para visualizar el modelo

```

<launch>
  <arg name="model" />
  <arg name="gui" default="False" />
  <param name="robot_description" textfile="$(arg model)" />
  <param name="use_gui" value="$(arg gui)" />
  <node name="estados_robot_pub" pkg="robot_state_publisher" type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find

```

```
tfm_simulacion)/rviz/tfm_simulacion.rviz" required="true" />
</launch>
```

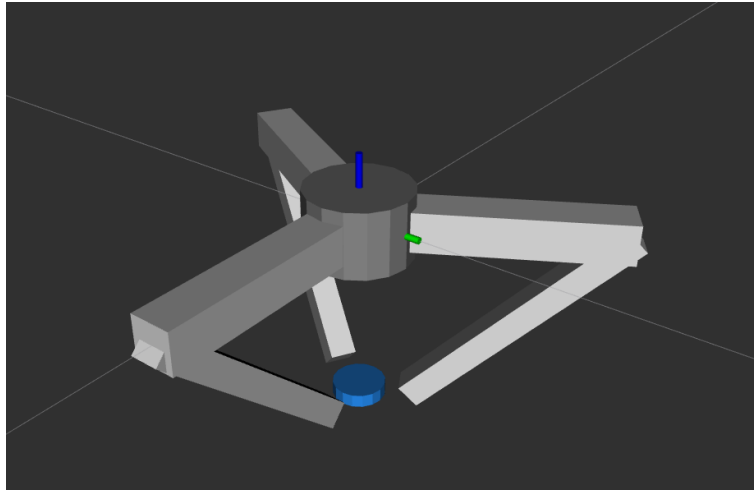


Figura 52: Modelo URDF del robot

Para poder gobernar el modelo y que todas las articulaciones se muevan de forma adecuada debemos además obtener los ángulos de rotación de las articulaciones que hemos definido en el modelo *URDF* del robot.

En la figura 53 vemos como se define el primer de estos ángulos, α , como la rotación en torno al eje Y partiendo desde una posición horizontal (paralela al eje X) del brazo inferior. Definida la geometría podemos obtener fácilmente α con la expresión 17.

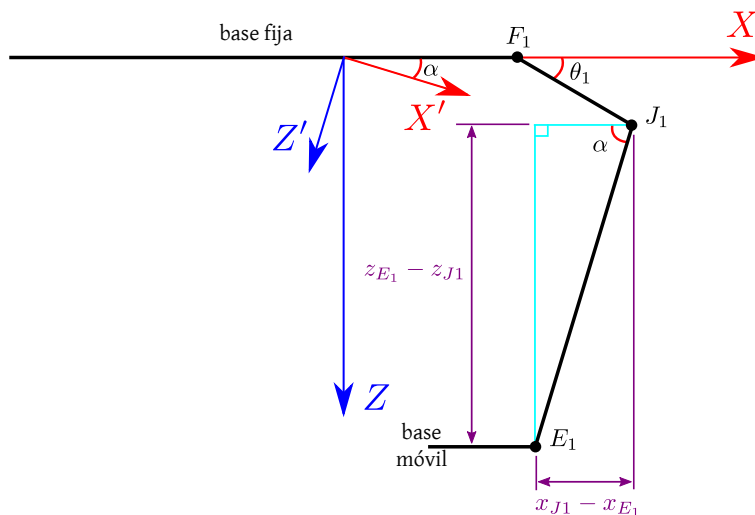


Figura 53: Definición de α y rotación para obtener β

$$\alpha = \arctan \frac{z_{E_1} - z_{J_1}}{x_{J_1} - x_{E_1}} \quad (17)$$

De la rotación de los ejes XYZ anteriores en torno al eje Y con el ángulo α resultan los ejes $X'Y'Z'$. Con esto podemos ver la definición del segundo de los ángulos, β , como la rotación en torno al eje Z' del codo para que el final del brazo inferior del robot alcance su posición real. En la figura 54 vemos un esquema de la geometría que interviene en su cálculo, que se realiza mediante la sencilla expresión de la ecuación 18. Los puntos del codo y la unión

del brazo al actuador (J_1 y E_1) deben rotarse convenientemente para dar lugar a los puntos J'_1 y E'_1 .

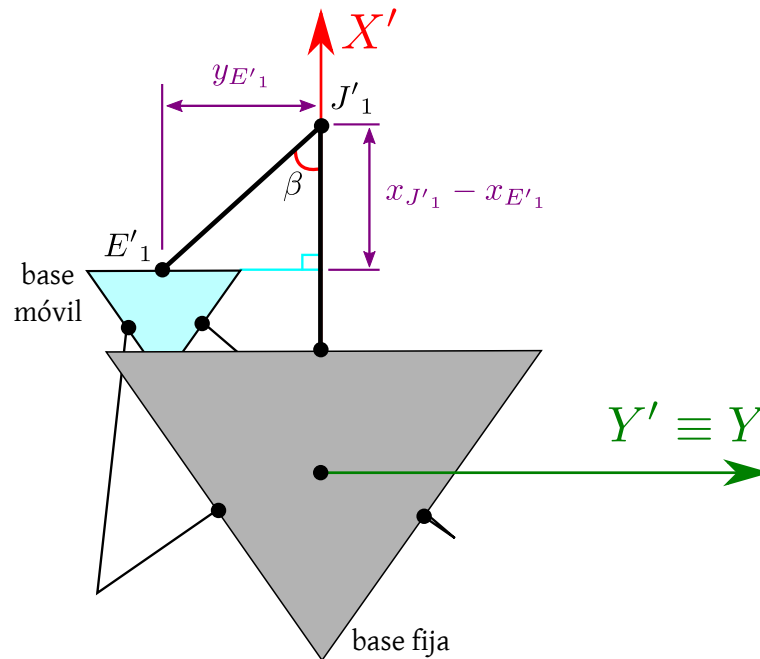


Figura 54: Definición de β

$$\beta = \arctan \frac{y_{E'_1}}{x_{J'_1} - x_{E'_1}} \quad (18)$$

El código de Python del listado de código 20 se encarga de la tarea de calcular estos dos ángulos en cada uno de los tres codos del robot (E_1 , E_2 y E_3). En este listado vemos varias variables que definen la geometría específica de nuestro robot y varios puntos de interés, una serie de funciones auxiliares de apoyo (funciones de rotación, suma de vectores...) y finalmente una función (*angulos_codo*) que calcula los dos ángulos de las articulaciones del codo de un brazo específico partiendo de la información de la posición del actuador y del brazo con el que trabaja.

Listado de código 20 Python que calcula la posición de las articulaciones del modelo en URDF

```
import math

# Specific geometry for my delta robot:
e = 130.0 # small triangle side (EE)100
f = 125.0 # support triangle side215
re = 205.0 # upper arm length350
rf = 150.0 # lower arm length250
hf = math.sqrt(0.75*(f**2))
he = math.sqrt(0.75*(e**2))

dtr = math.pi / 180.0 # degrees to radians

# Initial points
A1 = [hf/3, 0, 0]
A2 = [-A1[0]*math.cos(60*dtr), -A1[0]*math.sin(60*dtr), 0]
A3 = [A2[0], -A2[1], 0]
```

```
def punto_codo(theta):
    theta *= dtr
    b1 = [A1[0]+rf*math.cos(theta), 0.0, rf*math.sin(theta)]
    return b1

sin120 = math.sin(120*dtr)
cos120 = math.cos(120*dtr)
def rotacion120(ent):
    sal = [0.0, 0.0, 0.0]
    sal[0] = cos120*ent[0]+sin120*ent[1]
    sal[1] = -sin120*ent[0]+cos120*ent[1]
    sal[2] = ent[2]
    return sal

sin240 = math.sin(240*dtr)
cos240 = math.cos(240*dtr)
def rotacion240(ent):
    sal = [0.0, 0.0, 0.0]
    sal[0] = cos240*ent[0]+sin240*ent[1]
    sal[1] = -sin240*ent[0]+cos240*ent[1]
    sal[2] = ent[2]
    return sal

def sumav(v1, v2):
    s = [0.0, 0.0, 0.0]
    s[0] = v1[0] + v2[0]
    s[1] = v1[1] + v2[1]
    s[2] = v1[2] + v2[2]
    return s

vhe1 = [he/3, 0.0, 0.0]
vhe2 = rotacion120(vhe1)
vhe3 = rotacion120(vhe2)
def punto_ee(ee, brazo):
    sal = [0.0, 0.0, 0.0]
    if brazo == 1:
        sal = sumav(ee, vhe1)
    elif brazo == 2:
        sal = sumav(ee, vhe2)
    elif brazo == 3:
        sal = sumav(ee, vhe3)
    return sal

def rotacion_y(ent, ang):
    sal = [0.0, 0.0, 0.0]
    sal[0] = ent[0]*math.cos(ang) - ent[2]*math.sin(ang)
    sal[1] = ent[1]
    sal[2] = -ent[0]*math.sin(ang) + ent[2]*math.cos(ang)
    return sal

def angulos_codo(codo, ee, brazo):
    if brazo == 2:
        codo = rotacion240(codo)
        ee = rotacion240(ee)
    elif brazo == 3:
        codo = rotacion120(codo)
        ee = rotacion120(ee)
    if (codo[0]-ee[0]) != 0:
```

```
        ang_a = math.atan((ee[2]-codo[2])/(codo[0]-ee[0]))
    else:
        ang_a = 1.570796326794897
    # prep
    codo = rotacion_y(codo, ang_a)
    ee = rotacion_y(ee, ang_a)
    # calc
    if (codo[0]-ee[0]) != 0:
        ang_b = math.atan((ee[1])/(codo[0]-ee[0]))
    else:
        ang_b = 0
    return [ang_a, ang_b]
```

Con esto y la funciones de cinemática de la sección 3 ya podemos colocar todas las articulaciones del modelo en función de los ángulos de los tres brazos. Para mover el modelo de esta forma debemos comunicarnos con rviz a través de ROS usando los mensajes adecuados, emplearemos un mensaje predefinido de ROS, *JointState*, y otro definido en nuestro paquete de visualización, *angulos* (listado de código 21).

Listado de código 21 Nuevo mensaje definido para el paquete de visualización

```
float32 theta1
float32 theta2
float32 theta3
```

El objetivo es que un nuevo nodo se encargue de escuchar los ángulos que se publiquen y calcule y publique las posiciones de todas articulaciones de forma adecuada para mover el modelo. Este nodo se muestra en el listado de código 22

Listado de código 22 Código Python que maneja el nodo que mueve el modelo del robot

```
#!/usr/bin/env python
import delta_kinematics
import codos
import roslib
import rospy
import std_msgs.msg
from sensor_msgs.msg import JointState
from tfm_simulacion.msg import angulos

pi = 3.141592653
dtr = pi / 180.
joint = JointState()

def callback(data):
    global joint
    # rospy.loginfo('callback')
    p = delta_kinematics.forward(data.theta1, data.theta2, data.theta3)
    punto = [p[1], p[2], p[3]]
    c1 = codos.punto_codo(data.theta1)
    p1 = codos.punto_ee(punto, 1)
    [a1_a, a1_b] = codos.angulos_codo(c1, p1, 1)
    c2 = codos.punto_codo(data.theta2)
    c2 = codos.rotacion120(c2)
    p2 = codos.punto_ee(punto, 2)
    [a2_a, a2_b] = codos.angulos_codo(c2, p2, 2)
    c3 = codos.punto_codo(data.theta3)
    c3 = codos.rotacion120(c3)
    c3 = codos.rotacion120(c3)
```



```
p3 = codos.punto_ee(punto, 3)
[a3_a, a3_b] = codos.angulos_codo(c3, p3, 3)
# Datos para publicar
joint.header = std_msgs.msg.Header()
joint.header.stamp = rospy.Time.now()
joint.name = ['base_brazo1', 'base_brazo2', 'base_brazo3', 'codo1_a',
'codo1_b', 'codo2_a', 'codo2_b', 'codo3_a', 'codo3_b', 'act_x', 'act_y',
'act_z']
joint.position = [data.theta1 * dtr, data.theta2 * dtr, data.theta3 *
dtr, data.theta1 * dtr + a1_a, -a1_b, data.theta2 * dtr + a2_a, -a2_b,
data.theta3 * dtr + a3_a, -a3_b, punto[0] / 1000, -punto[1] / 1000,
punto[2] / 1000]
joint.velocity = []
joint.effort = []

def nodo():
    pub = rospy.Publisher('joint_states', JointState, queue_size=10)
    rospy.init_node('posicionador_modelo', anonymous=False)
    rate = rospy.Rate(7.8125) #Hz
    while not rospy.is_shutdown():
        rospy.Subscriber("angulos_delta", angulos, callback) # listener
        pub.publish(joint)
        rate.sleep()

if __name__ == '__main__':
    try:
        nodo()

    except rospy.ROSInterruptException:
        pass
```

Vamos a diseccionar el código y explicar su funcionamiento:

- Importa las funciones de la cinemática del robot, las funciones definidas en el listado de código 20 explicadas anteriormente y las librerías y mensajes de ROS.
- Define variables geométricas e inicializa un mensaje de tipo *JointState*.
- Función "callback" que será llamada cada vez que se recibe una nueva posición de los tres motores ($\theta_1, \theta_2, \theta_3$) y que calcula el estado de todas las articulaciones correspondientes a esa nueva posición (utilizando la cinemática del robot y las funciones de *codos*). Todos los estados junto con un encabezamiento de tiempo será el mensaje que publicaremos para posicionar el modelo en la visualización de RVIZ.
- La función *nodo* inicializa el nodo, especificando el tema donde se va a publicar y el tipo de mensaje. Se especifica la frecuencia a la que se publicarán el estado de las articulaciones del robot. Al igual que para el robot real se va a publicar a 7,8125 Hz.
- Llamada a la función *nodo* con la estructura habitual en Python.

Ejecutando este nodo y publicando una posición válida en el tema de ROS *angulos_delta* el modelo se posicionará en la posición indicada. Por ejemplo si le pedimos que se coloque en la configuración correspondiente para los ángulos $(\theta_1, \theta_2, \theta_3) = (71^\circ, 45^\circ, 35^\circ)$ obtendremos la visualización mostrada en la figura 55.

Finalmente, para conseguir una visualización simultanea al movimiento del robot, es necesario crear un nodo más que se subscriba a los pulsos de los motores al igual que el nodo

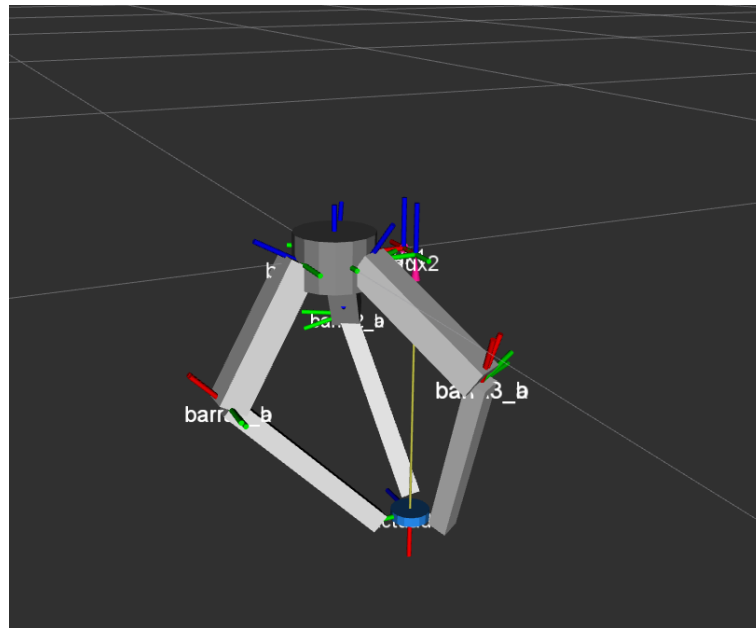


Figura 55: move

de Arduino y que reproduzca el movimiento. Este nodo es el que se muestra en el listado de código 23.

Listado de código 23 Código Python que maneja el nodo que convierte los trenes de pulsos en posiciones de los motores para crear un visualización simultánea al movimiento real del robot

```
#!/usr/bin/env python

import rospy
from tfm_simulacion.msg import angulos
from tfm_delta.msg import Pulsos

pulsos_recibidos = Pulsos()
angulos_envio = angulos()
angulos_envio.theta1 = 0.0
angulos_envio.theta2 = 0.0
angulos_envio.theta3 = 0.0
step = 1.8/8
i = 0
nuevo = 0
hecho = 0

def nopulses():
    pulsos_recibidos.id = int(0)
    pulsos_recibidos.p1f = int(0)
    pulsos_recibidos.p1b = int(0)
    pulsos_recibidos.p2f = int(0)
    pulsos_recibidos.p2b = int(0)
    pulsos_recibidos.p3f = int(0)
    pulsos_recibidos.p3b = int(0)

def actualiza(data):
    global pulsos_recibidos
    global hecho
```

```
pulsos_recibidos = data
hecho = 0

def principal():
    rospy.init_node('pulsos_a_angulos', anonymous=False)
    rate = rospy.Rate(62.5) # 62.5 Hz, T = 16 ms.
    rospy.Subscriber("trenes_pulsos", Pulsos, actualiza)
    pub = rospy.Publisher('angulos_delta', angulos, queue_size=10)
    nopulses()
    i = 0
    step = 0.225
    global hecho
    hecho = 1
    while not rospy.is_shutdown():
        if hecho == 0:
            if pulsos_recibidos.p1f >> i & 0b1:
                angulos_envio.theta1 += step
            if pulsos_recibidos.p1b >> i & 0b1:
                angulos_envio.theta1 -= step
            if pulsos_recibidos.p2f >> i & 0b1:
                angulos_envio.theta2 += step
            if pulsos_recibidos.p2b >> i & 0b1:
                angulos_envio.theta2 -= step
            if pulsos_recibidos.p3f >> i & 0b1:
                angulos_envio.theta3 += step
            if pulsos_recibidos.p3b >> i & 0b1:
                angulos_envio.theta3 -= step
            i += 1
            if i >= 8:
                i = 0
                hecho = 1
        pub.publish(angulos_envio)
        rate.sleep()

if __name__ == '__main__':
    try:
        principal()

    except rospy.ROSInterruptException:
        pass
```

En el código encontramos los siguientes puntos destacables:

- Importación de las bibliotecas de ROS y los mensajes que serán necesarios.
- Inicialización de todas las variables y constantes: los pulsos que se reciben, la posición angular resultante que será publicada, el paso del motor con cada pulso que reciba y variables de control de flujo.
- Función *nopulses()*: Se utiliza para resetear más rápidamente la variable que contiene los pulsos que se reciben.
- Función *actualiza()*: Función asociada a la suscripción al tema *trenes_pulsos* y que actualiza la variable con los pulsos recibidos y resetea una variable de flujo.
- Función *principal()*: Inicializa el nodo y realiza acciones paralelas al nodo de Arduino que mueve el robot real. Se suscribe al tema *trenes_pulsos* para recorrerlos e ir actualizando de forma sincronizada la posición de los motores que publicará al tema

angulos_delta que es el que marca la configuración de la visualización como hemos visto antes. El periodo es igual al del nodo de Arduino ($T = 16\text{ ms}$). Se define el incremento angular que supone cada paso ($step = 0,225^\circ$) y el resto del funcionamiento es igual al que se explico en la sección de Arduino anteriormente solo que esta vez utilizando el lenguaje Python.

- Llamada a la función *principal()* con la estructura habitual de Python.

Con esto la visualización está lista para realizar un movimiento de forma simultanea al movimiento del robot real.

7. Resultados

En esta sección vamos a mostrar los resultados obtenidos al poner a prueba el robot y se va a comentar los problemas encontrados en la ejecución de los dibujos y en el funcionamiento del robot delta.

En primer lugar se realizan pruebas con dibujos sencillos hechos a mano y convertidos en formato SVG. Comenzando por una serie de figuras ovaloides decrecientes, figura 56a. Vemos que el resultado no es del todo preciso: las formas ovaloides más grandes no se cierran, según se hacen pequeñas pierden la forma y que la más pequeña de todas los trazos están superpuestos en tan pequeña dimensión.

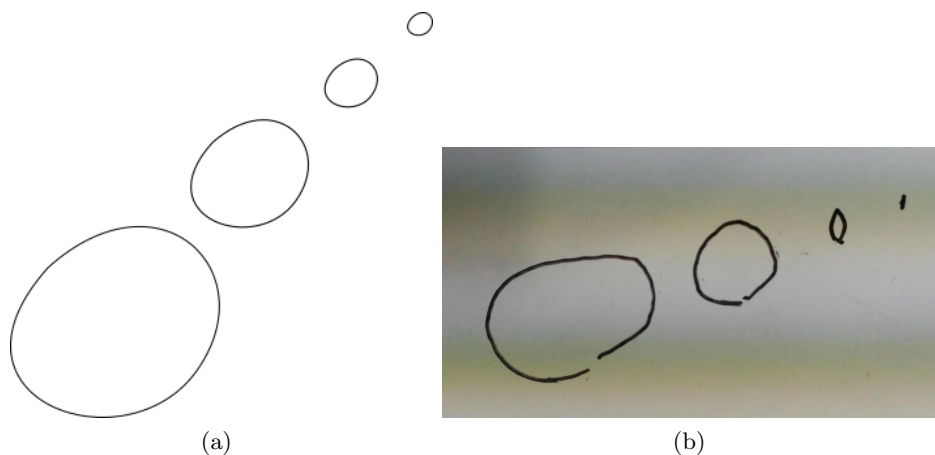


Figura 56: Dibujo con figuras ovaloides. Formato digital (a) y dibujo del robot (b)

La segunda prueba consiste en dos figuras una cerrada y triangular y otra abierta, con salientes y cambios de trazos concavos a convexos, figura 57a. En el resultado, figura 57b, observamos que las figuras salen algo deformadas pero consigue cerrar correctamente la figura triangular y dejar abierta la otra. Además, algunos de los angulos agresivos en el trazo no se consiguen perfectamente.

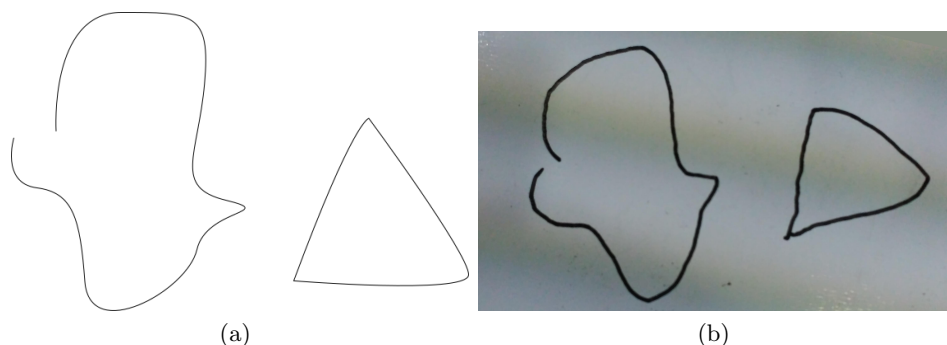


Figura 57: Dibujo con figuras ovaloides. Formato digital (a) y dibujo del robot (b)

El siguiente test es una cara muy básica dibujada a mano alzada y convertida en SVG, figura 58a. El resultado, figura 58b es bastante reconocible, sin embargo podemos ver como los ojos no se cierran y el trazo exterior que delimita la cara se cierra deficientemente, arrastra el rotulador un poco más de lo necesario sobre la pizarra al cambiar de un trazo al siguiente, es decir no se levanta lo suficiente antes de comenzar a desplazarse hacia la siguiente forma a dibujar.

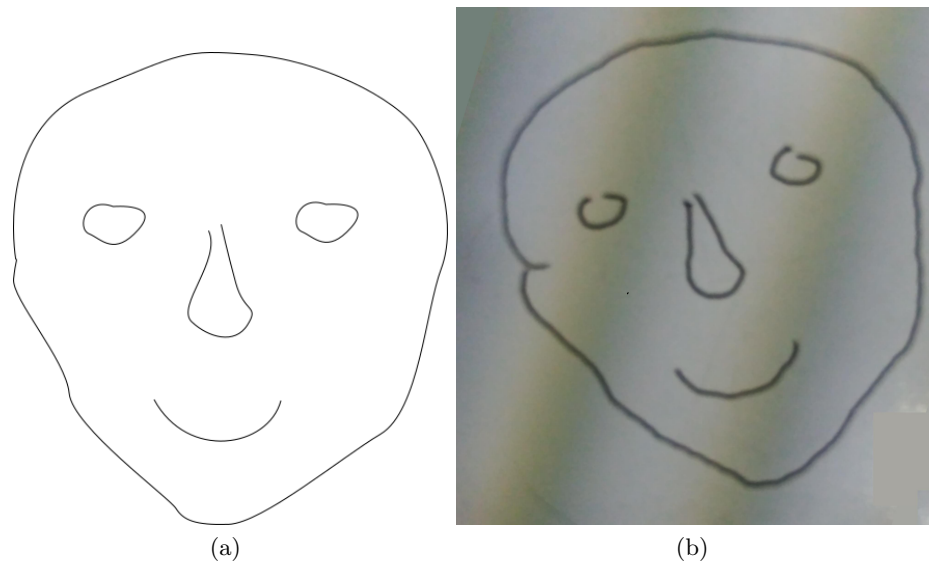


Figura 58: Dibujo con una cara sencilla. Formato digital (a) y dibujo del robot (b)

Se prueba a continuación con una imagen sencilla, un dibujo simple de la cara de un niño, originalmente en formato JPG (figura 59a). La imagen se convierte en formato SVG a través de Potrace como se expuso en la Sección 5. Luego el robot dibuja igual que en los casos anteriores, el resultado se muestra en la figura 59b.

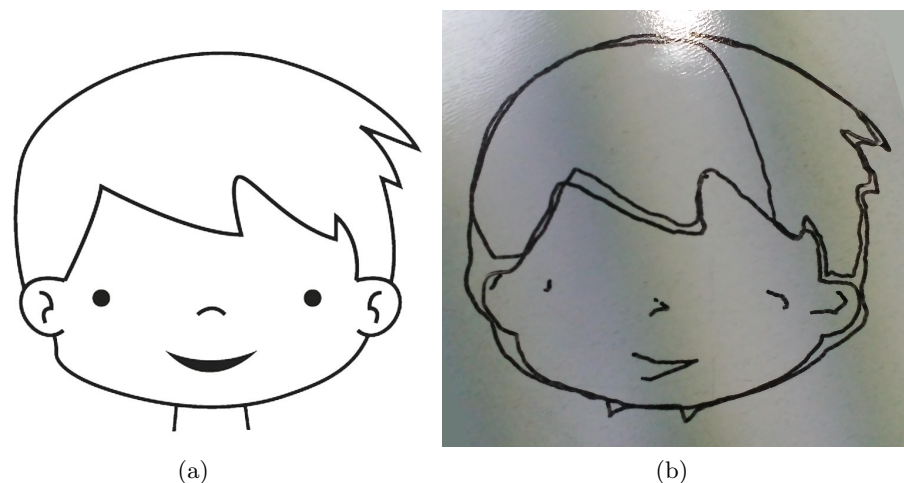


Figura 59: Dibujo con una cara sencilla. Formato digital (a) y dibujo del robot (b)

El dibujo es reconocible pero podemos ver varios errores claros en el mismo: los trazos externos, que en la foto original son claramente una única curva, están duplicados; los trazos pequeños (orejas, nariz, ojos...) no se consiguen dibujar bien con la forma original, más bien, se observa que el actuador se a posicionado correctamente pero parece no haberse movido por todo el trazo; y un gran trazo atraviesa toda el pelo del niño del dibujo.

A la vista de los resultados, podemos afirmar que los dibujos no son ni mucho menos perfectos, observamos diversos fallos que deben ser corregidos. A modo de comentario para conseguir mejorar futuras se va a justificar la probable causa de algunos de estos fallos:

- En los trazos más pequeños y detalles a veces ocurre que el rotulador no resvala correctamente, a pesar de que el robot se mueve el rotulador queda un poco atascado en

su posición y el robot no realiza la suficiente fuerza o el suficiente desplazamiento para moverlo como se desearía.

- En algunos dibujos los trazos cerrados no se llegan a completar, eso es debido a que el robot se levanta para cambiar de trazo antes de tiempo.
- En contraposición a este último punto, en otras ocasiones al acabar un trazo el robot sigue dibujando un poco más antes de abandonar totalmente el plano de trabajo para dirigirse al siguiente trazo, es decir, el robot se levanta tarde en el cambio de trazo.
- Cuando Potrace realiza la conversión a SVG convierte correctamente las figuras rellenas. Sin embargo, las que son simplemente trazos y curvas las entiende también como figuras rellenas, esto provoca que las recorra dos veces (una por cada lado, como siguiendo su perímetro). Este inconveniente se observa claramente en la figura 60 que corresponde a la trayectoria diseñada para el dibujo de la figura 59.

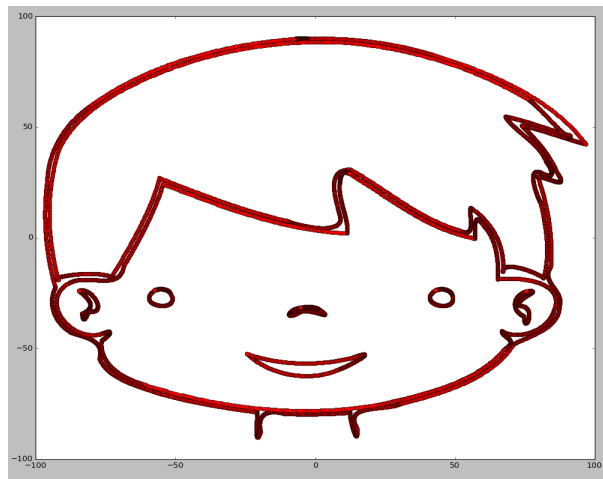


Figura 60: Trazados duplicados debido a la conversión de Potrace

- El juego que puedan tener las distintas articulaciones del robot juega por supuesto un papel relevante en la exactitud del movimiento que realiza el mismo, pudiendo ser causa de algunas deformaciones en las figuras y de trazos poco exactos. Este juego puede ser causado por un mal ajuste pero también por la poca rigidez que tienen las piezas de plástico empleadas en comparación con la que ofrecerían unas metálicas (esto era conocido, pero dado el carácter didáctico del proyecto como se ha indicado al comienzo, se ha dejado esta circunstancia en un segundo plano).

8. Planificación temporal y costes

En este apartado se describe la planificación temporal llevada a cabo y se valoran los medios materiales y humanos empleados para realizar el presente trabajo, con el fin de realizar una estimación del coste económico del mismo.

8.1. Planificación temporal

Para describir la planificación llevada a cabo se va a indicar primero las tareas básicas que se han realizado en este trabajo:

- **Diseño mecánico:** Engloba todas las tareas de diseño mecánico del robot: decisiones de diseño, diseño de las piezas para impresión 3D, selección de componentes, rediseño, etc.
- **Frabricación, montaje y ensamblaje:** Todas las tareas que se llevan a cabo para trasladar el diseño sobre el papel del robot a la realidad: impresión de las piezas de plásticos, construcción de los elementos que formarán el robot, ensamblaje de piezas intermedias y montaje final del robot.
- **Diseño del control:** Entran en este punto tareas tanto de planificación como de diseño de la forma de mover el robot, como son: elección del metodo de control, diseño de su funcionamiento, planificación de las conexiones, diseño de la comunicación...
- **Programación de los algoritmos:** Transformar el motodo de control en un método efectivo y que funcione requiere todas las tareas de este punto: desarrollo de scripts, programación de nodos, creación del modelo virtual del robot, programación del control de Arduino, pruebas de funcionamiento, etc.
- **Presentación de los resultados:** Se incluye aquí la redacción de la presente documentación y todas las tareas asociadas.

Se presenta ahora una aproximación en forma de diagrama de Gantt de la planificación seguida para realizar este trabajo.

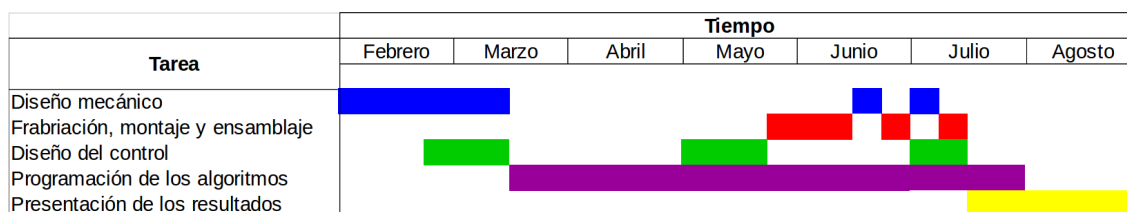


Figura 61: Diagrama de Gantt

8.2. Costes

Estimamos ahora los costes del proyecto, para ello se realiza un presupuesto en el que distinguimos tres categorías generales para los gastos realizados:

1. **Componentes:** Incluye todos los gastos provocados por la adquisición de materiales y elementos necesarios para la fabricación del robot delta descrito en este proyecto.
2. **Consumos:** Contiene todos los gastos debidos a la realización del proyecto y el uso del laboratorio de Sistemas distribuidos de Control.

Tanto el gasto en electricidad como el gasto en la conexión a internet son calculados de manera aproximada, ya que no se dispone de acceso a los datos exactos.

Por otro lado, la última partida (Material reutilizado y amortización de los equipos) trata de reflejar los gastos debidos a la utilización de material presente en el laboratorio para llevar a cabo el montaje del robot como los gastos que implican el uso de equipos (como el ordenador o la impresora 3D) en forma de pérdida de valor por la amortización acumulada durante el período en que ha transcurrido el proyecto. Se trata también un valor aproximado por no conocer el valor real de los activos.

3. **Mano de obra:** Plasma el coste que supondría el desarrollo de este proyecto por un ingeniero que se encargase de realizar las mismas tareas descritas en este proyecto, es decir las tareas descritas en la planificación temporal.

El detalle del presupuesto se presenta a continuación:

(Nota: El IVA está incluido en todos los precios.)

Nº de Orden	Concepto	Unidades	Cantidad	Precio unitario (€/ud.)	Precio total (€)
1	Componentes				
1.1	Motores paso a paso	ud.	3	33.72	101.16
1.2	Adafruit Motorshield V2	ud.	2	26.93	53.86
1.3	Driver HY-DIV268N-5A	ud.	2	20.60	41.20
1.4	Driver DQ420MA	ud.	1	25.19	25.19
1.5	Fuente de 12 V	ud.	1	15.95	15.95
1.6	Tornillería	-	-	10.00	10.00
1.7	Cables diversos	m	5	0.40	2.00
1.8	Bobina de plástico ABS para impresión 3D	ud.	1	15.77	15.77
1.9	Rotulador BIC Velleda	ud.	2	0.90	1.80
2	Consumos				
2.1	Electricidad	kWh	3000	0.13	382.50
2.2	Conexión a internet	h	500	0.07	35.00
2.3	Material reutilizado y amortización de equipos	-	-	200.00	200.00
3	Mano de obra				
3.1	Ingeniero. Diseño mecánico y del control	h	150	25.00	3750.00
3.2	Ingeniero. Programación de los algoritmos	h	300	20.00	6000.00
3.3	Ingeniero. Fabricación, montaje y ensamblaje	h	100	15.00	1500.00
3.4	Ingeniero. Presentación de los resultados	h	100	15.00	1500.00
TOTAL					13.634,43 €

Por lo que el presupuesto asciende a la valoración total (IVA incluido) de trece mil seis cientos treinta y cuatro coma cuarenta y tres Euros (13.634,43€).

Lois Rilo Antelo
45846316W

En Barcelona, a 5 de Septiembre de 2016

9. Conclusiones

En este trabajo se ha desarrollado un robot delta capaz de realizar una representación gráfica a partir de imágenes digitales partiendo de cero. Se ha pasado por todas las etapas de diseño: desde el análisis de la cinemática del robot, la planificación del diseño conforme a diversos condicionantes (finalidad, herramientas y materiales disponibles, coste...) y la selección de componentes (motores paso a paso, controladoras...), hasta el diseño de piezas nuevas para el robot y el rediseño ante malas elecciones. Con todo diseñado se han fabricado y reunido todos los componentes necesarios y se ha construido el robot.

Paralelamente, se ha desarrollado un método para la extracción de la información de imágenes digitales para su conversión en trayectorias que pudiese realizar el robot. Estas trayectorias se han traducido a trenes de pulsos, que es la forma elegida para comunicarse con el robot y provocar su movimiento.

Se ha empleado ROS para realizar la comunicación entre ordenador y robot, mientras que Arduino ha sido el microcontrolador encargado de administrar y ejecutar los trenes de pulsos. Aprovechando las herramientas que proporciona ROS también se ha creado un paquete de visualización del movimiento del robot con RVIZ y un modelo en URDF.

Se termina mostrando los resultados obtenidos y capacidades del robot, así como sus carencias, tratando de reflejar las razones de algunas de estas e indicando donde concentrar esfuerzos futuros.

Finalmente toca analizar las motivaciones y objetivos iniciales. Este trabajo ha obligado al alumno a explorar una serie de campos y adquirir ciertas habilidades que motivaban la realización del mismo en un principio. Asimismo todos los objetivos del trabajo se han cumplido en gran medida, hasta el punto de que finalmente se ha construido un robot que realmente era capaz de dibujar.

Ya solo queda añadir, que será el tiempo quien se encargará de decidir si las habilidades y conocimientos adquiridos, el acercamiento a nuevos lenguajes de programación, al software libre, etc., son de utilidad en el futuro laboral del alumno.

Agradecimientos

Quiero expresar mi agradecimiento a la Universitat Politècnica de Catalunya por la formación de la que he disfrutado durante estos dos años de máster y que concluye con este trabajo.

También quiero dar gracias a mi director, Manel Velasco, por brindarme la oportunidad de trabajar en este robot en el laboratorio de Sistemas Distribuidos de Control, así como por la orientación que me ha dado.

Bibliografía

Referencias

- [1] Real Academia Española. (2016). *Diccionario de la lengua española*. Recuperado el 2 de Agosto de 2016 desde <http://dle.rae.es/?id=WYTm4uf>
- [2] Robotic Spot. (2011). *¿Qué es la robótica?*. Recuperado el 2 de Agosto de 2016 desde <http://www.roboticspot.com/robotica.php>
- [3] Boston Dynamics. (2016). *Atlas - The Agile Anthropomorphic Robot*. Recuperado el 2 de Agosto de 2016 desde http://www.bostondynamics.com/robot_Atlas.html
- [4] Rosell, J. (2016). *Service and Industrial Robotics*. Types of Robots. Barcelona.
- [5] Honeybee Robotics. (s.f.). *Honeybee on Mars*. Recuperado el 2 de Agosto de 2016 desde <http://www.honeybeerobotics.com/about-us/mars/>
- [6] Agencia EFE. (2015, 13 de Octubre). *Un grupo de ingenieros japoneses crea una prótesis robótica 'low cost'*. La Vanguardia. Recuperado el 2 de Agosto de 2016 desde <http://www.lavanguardia.com/salud/20151013/54437190186/ingenieros-japoneses-protesis-roboticas-low-cost.html>
- [7] Puerto, K. (2016, 18 de Marzo). *Así son los nuevos exoesqueletos de Panasonic, de todos los tamaños: Power Loader, Ninja y AWN-03*. Xataka. Recuperado el 2 de Agosto de 2016 desde <http://www.xataka.com/robotica-e-ia/panasonic-nos-ensena-sus-exoesqueletos-de-todos-los-tamanos-power-loader-ninja-y-awn-03>
- [8] Moreno, H. A. (2012). *Robots paralelos: Conceptos básicos y aplicaciones*. Centro de Automática y Robótica. Recuperado el 3 de Agosto de 2016 desde <http://es.slideshare.net/htrmoreno/robots-paralelos>
- [9] Clavel, R. (1990). *Device for the movement and positioning of an element in space*. EE.UU, Patente US4976582.
- [10] Clavel, R. (1991). *Conception d'un robot parallèle rapide à 4 degrés de liberté*. Thèse École polytechnique fédérale de Lausanne EPFL, n° 925.
- [11] Silva, L.A., Raguene, N., Saltaren, R., Sebastian, J.M. & Aracil, R. (2003). *Diseño, Simulación, Análisis Cinemático y Dinámico de un robot paralelo para Control Visual de altas prestaciones*. Universidad Politécnica de Madrid.
- [12] Inventables, Inc. *Stepper Motors*. Recuperado el 16 de Julio de 2016 desde <http://blog.inventables.com/p/stepper-motors.html>
- [13] Wikipedia. (s.f.). *Stepper Motor*. Recuperado el 16 de Julio de 2016 desde https://en.wikipedia.org/wiki/Stepper_motor
- [14] Earl, B. (2015, 23 de Noviembre). *What is a Stepper Motor?*. Adafruit. Recuperado el 16 de Julio de 2016 desde <https://learn.adafruit.com/all-about-stepper-motors/what-is-a-stepper-motor>
- [15] Reprap. (s.f.). *Stepper motor driver*. Recuperado el 17 de Julio de 2016 desde http://reprap.org/wiki/Stepper_motor_driver

- [16] Ada, L. (2016, 21 de Abril). *Adafruit Motor Shield V2 for Arduino. Overview*. Adafruit. Recuperado el 17 de Julio de 2016 desde <https://learn.adafruit.com/adafruit-motor-shield-v2-for-arduino/overview>
- [17] ThanksBuyer. (s.f.). *CNC Single Axis TB6600 0,2 – 5A Two Phase Hybrid Stepper Motor Driver Controller*. Recuperado el 18 de Julio de 2016 desde <http://www.thanksbuyer.com/cnc-single-axis-tb6600-0-2-5a-two-phase-hybrid-stepper-motor-driver-controller-24891>
- [18] Diotronic. (s.f.). *Driver para Motor NEMA 17*. Recuperado el 18 de Julio de 2016 desde http://www.diotronic.com/driver-para-motor-nema17_29358/

Bibliografía Complementaria

- *ROS tutorials* (s.f.). Recuperado el 1 de Agosto de 2016 desde <http://wiki.ros.org/ROS/Tutorials>
- *MathWorks Support* (s.f.). Recuperado el 1 de Agosto de 2016 desde http://es.mathworks.com/support/?s_tid=gn_supp
- González, J. (2015). *Diseño de piezas con Freecad*. Recuperado el 1 de Agosto de 2016 desde http://www.learobotics.com/wiki/index.php?title=Diseño_de_piezas_con_Freecad
- *FreeCAD help* (s.f.). Recuperado el 1 de Agosto de 2016 desde http://www.freecadweb.org/wiki/index.php?title=Online_Help_Toc/es
- *Curso de Python básico*. (2012). Recuperado el 1 de Agosto de 2016 desde <http://codigofacilito.com/cursos/Python>
- *Curso de C++ básico*. (2012). Recuperado el 1 de Agosto de 2016 desde <http://codigofacilito.com/courses/c-plus-plus>

Trabajo Fin de Máster
Máster Universitario de Ingeniería Industrial

**Diseño e implementación de un
sistema de control para la representación
gráfica a partir de imágenes**

ANEXO A: Planos

Autor: Lois Rilo Antelo
Director: Manel Velasco García
Convocatoria: Septiembre de 2016



Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona



ANEXO A: Planos

Plano N°1: Base superior

Plano N°2: Sujeción para motores

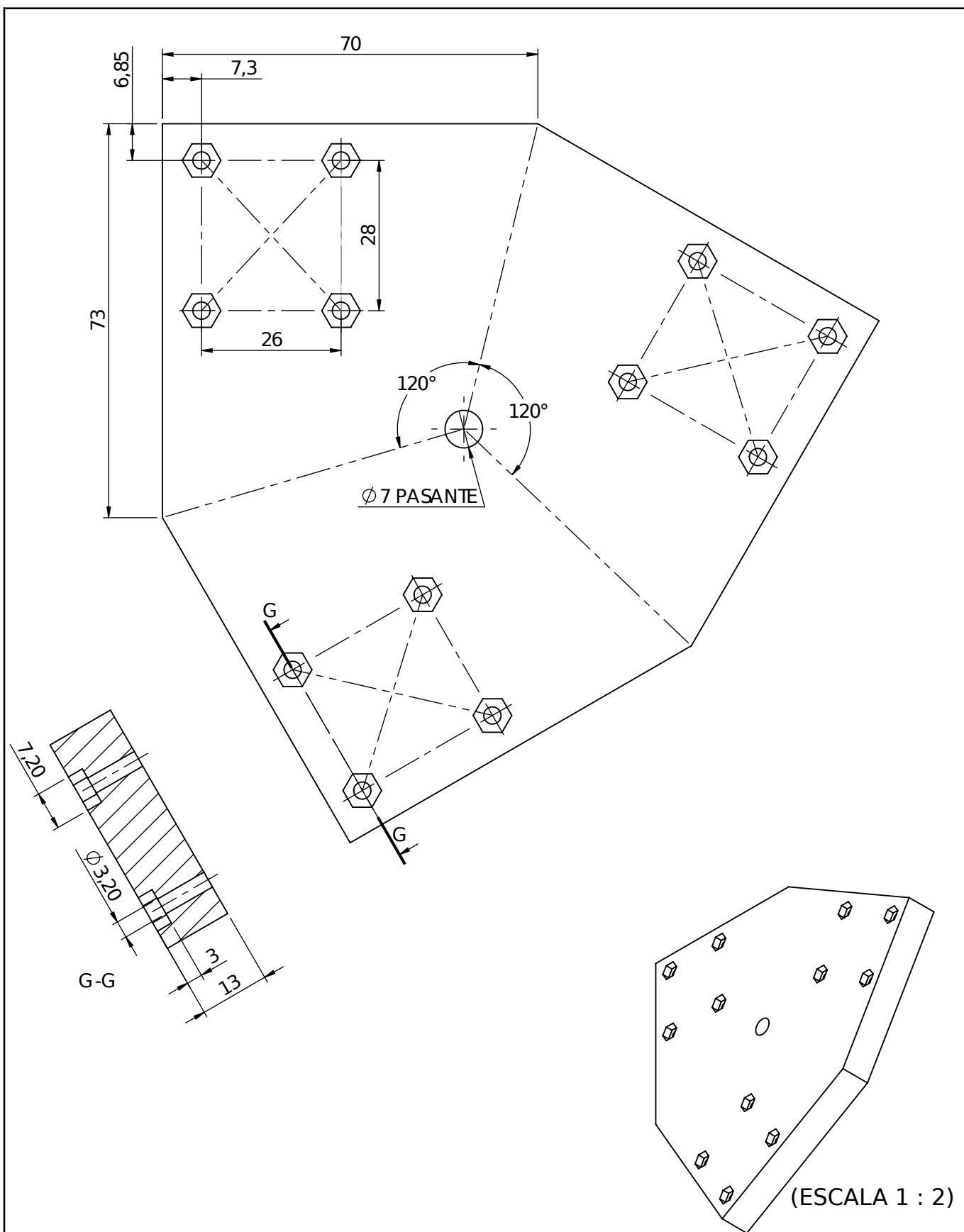
Plano N°3: Brazo superior

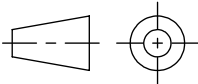

Plano N°4: Base móvil

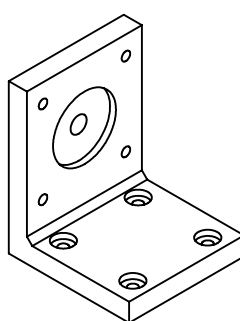
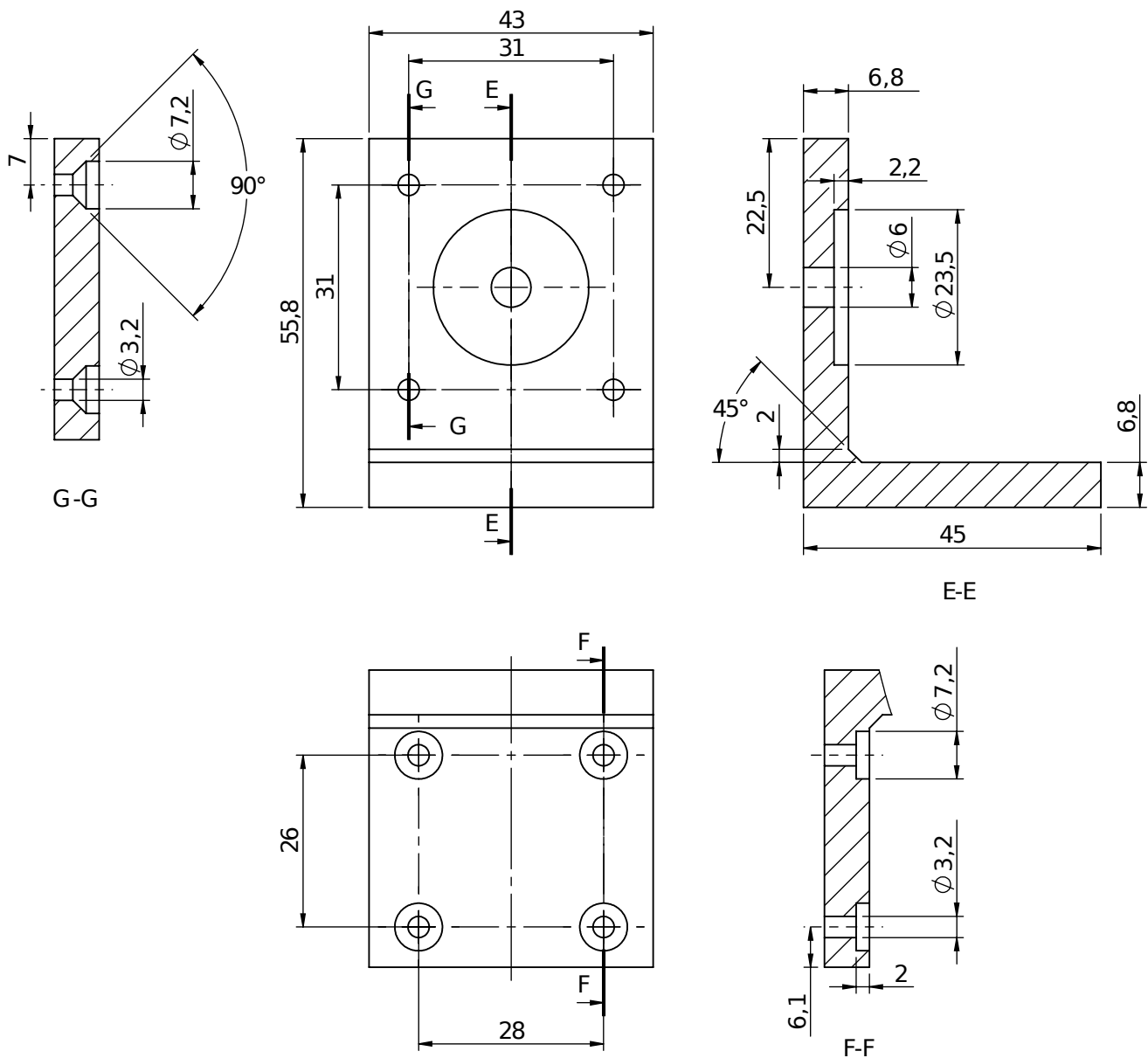
Plano N°5: Soporte actuador

Plano N°6: Pinza

Plano N°7: Soporte para controladoras

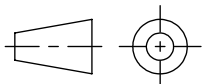
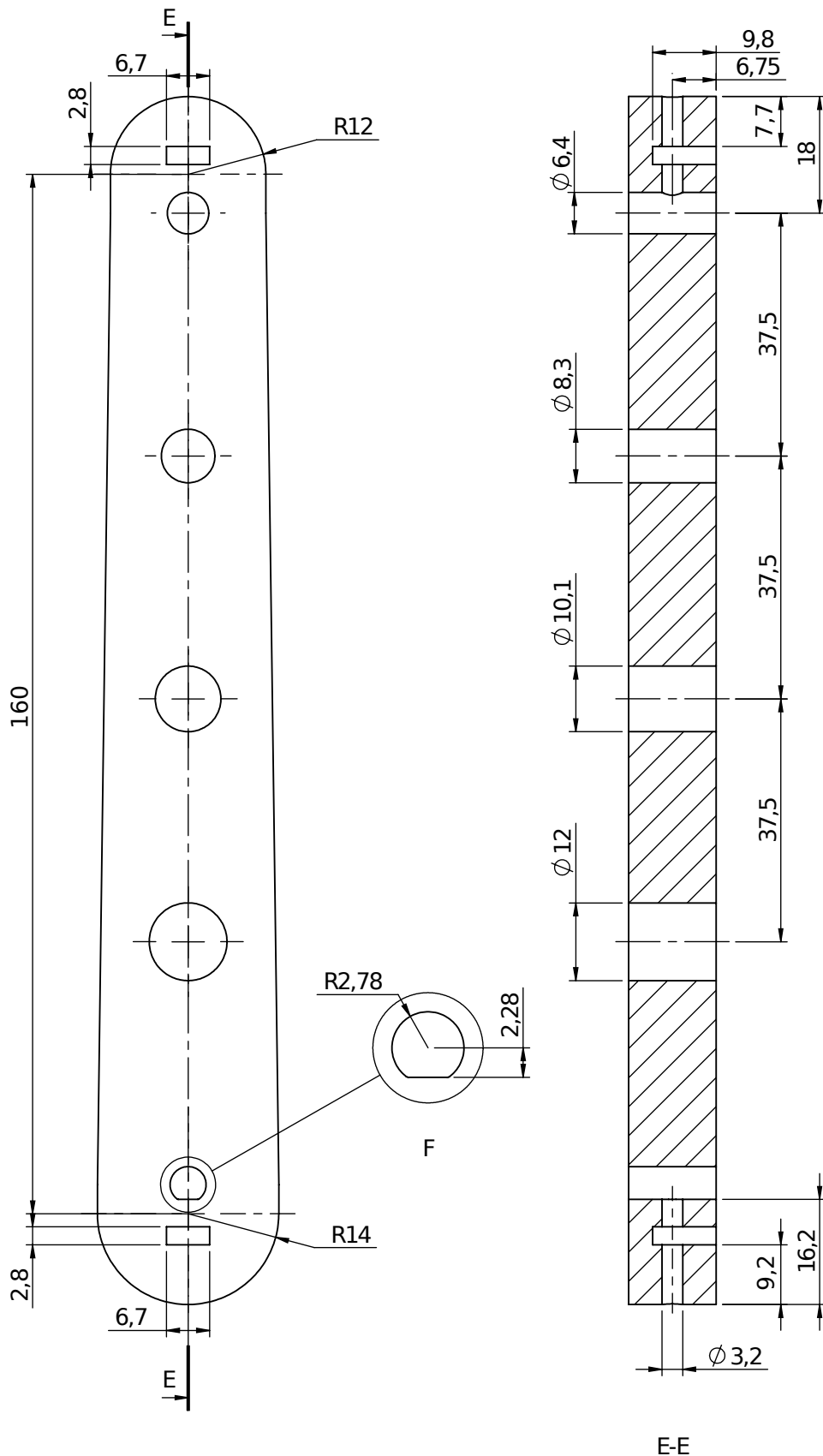


	<p>TFM:</p> <p>Diseño e implementación de un sistema de control para la representación gráfica a partir de imágenes</p>	<p>Autor:</p> <p>Lois Rilo Antelo</p>	<p>Material</p> <p>Plástico ABS</p>	<p>Escala</p> <p>1:1</p>
<p>Curso:</p> <p>2015/16</p>		<p>Denominación:</p> <p>Base superior</p>	<p>Plano:</p> <p>1 / 7</p> <p>Fecha:</p> <p>Agosto de 2016</p>	



(ESCALA 1 : 2)

		TFM: Diseño e implementación de un sistema de control para la representación gráfica a partir de imágenes	Autor: Lois Rilo Antelo	Material Plástico ABS	Escala 1:1
Curso: 2015/16			Denominación: Sujeción para motores	Plano: 2 / 7	Fecha: Agosto de 2016



TFM:

Diseño e implementación de
un sistema de control para
la representación gráfica a
partir de imágenes

Autor:

Lois Rilo Antelo

Denominación:

Brazo superior

Material

Plástico
ABS

Escala

1:1

Curso:
2015/16

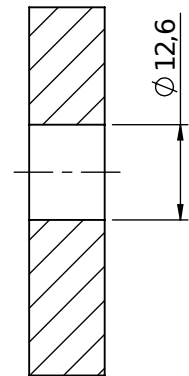
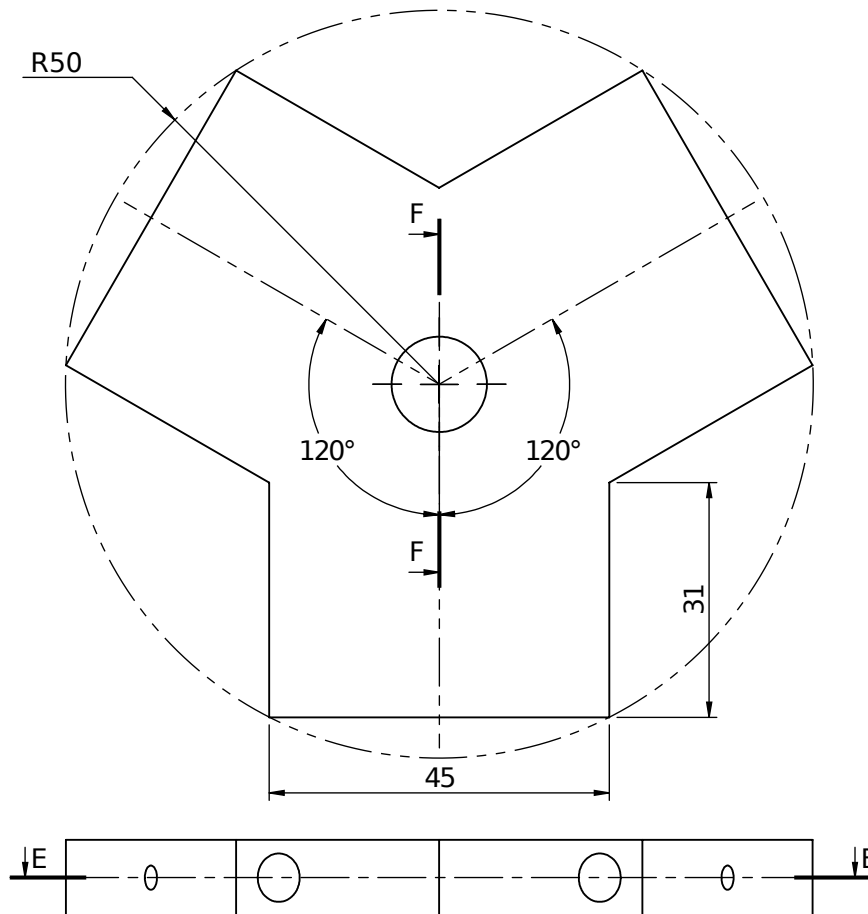


Plano:

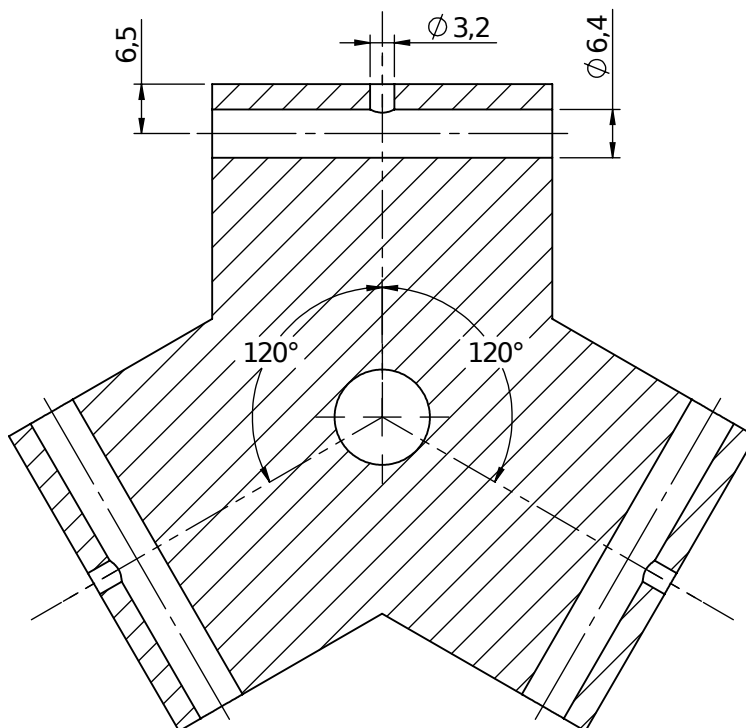
3 / 7

Fecha:

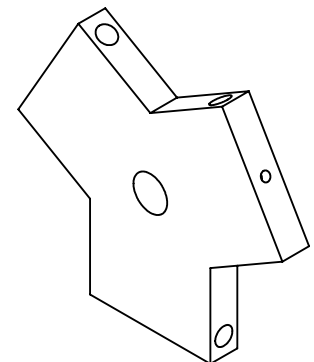
Agosto de 2016



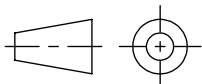
F-F



E-E



(ESCALA 1 : 2)



TFM:

Diseño e implementación de
un sistema de control para
la representación gráfica a
partir de imágenes

Autor:

Lois Rilo Antelo

Denominación:

Base móvil

Material

Plástico
ABS

Escala

1:1

Curso:
2015/16

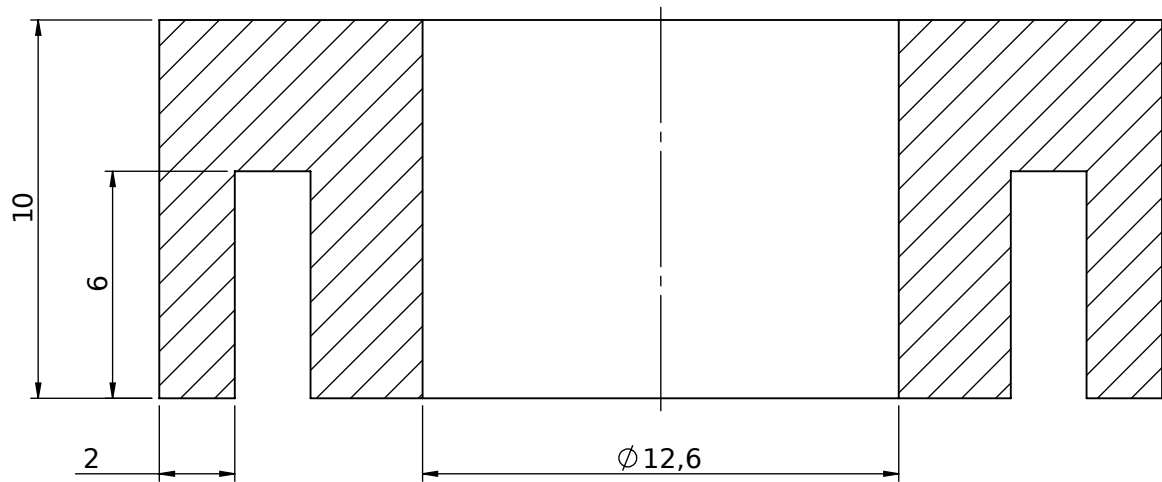
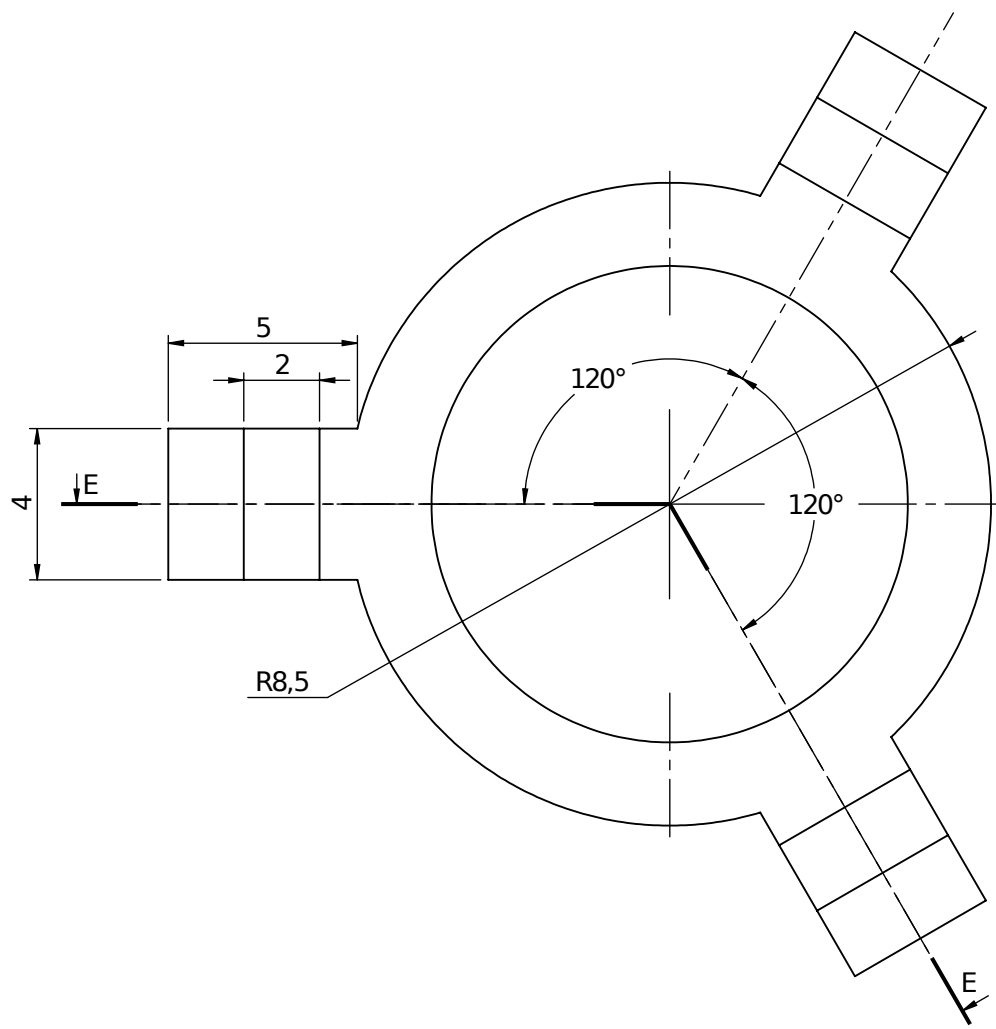


Plano:

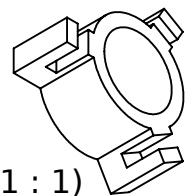
4 / 7

Fecha:

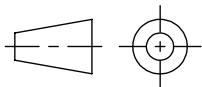
Agosto de 2016



E-E



(ESCALA 1 : 1)



TFM:

Diseño e implementación de
un sistema de control para
la representación gráfica a
partir de imágenes

Autor:

Lois Rilo Antelo

Denominación:

Soporte actuador

Material

Plástico
ABS

Escala

5:1

Curso:
2015/16

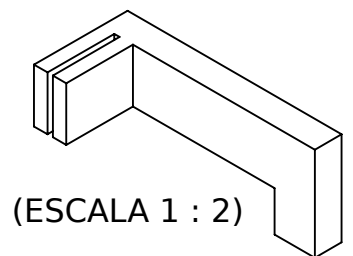
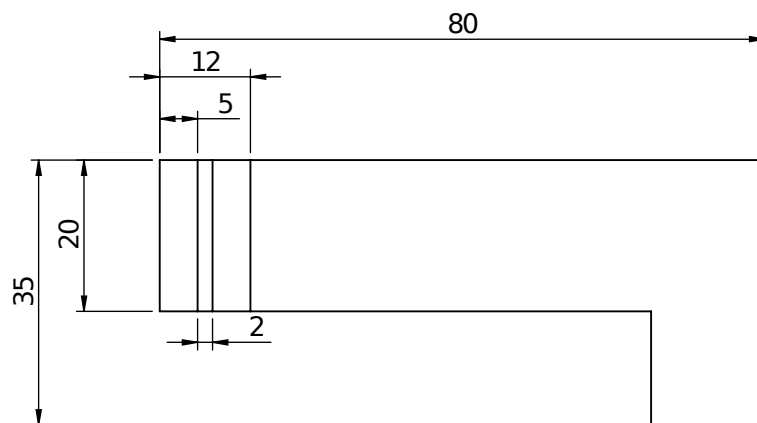
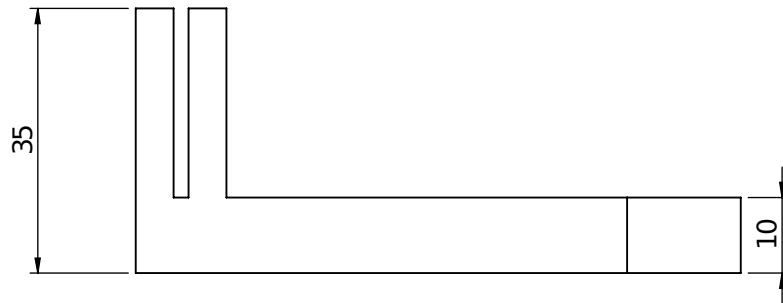


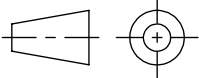

Plano:

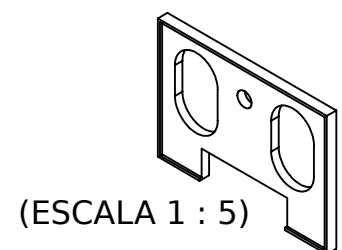
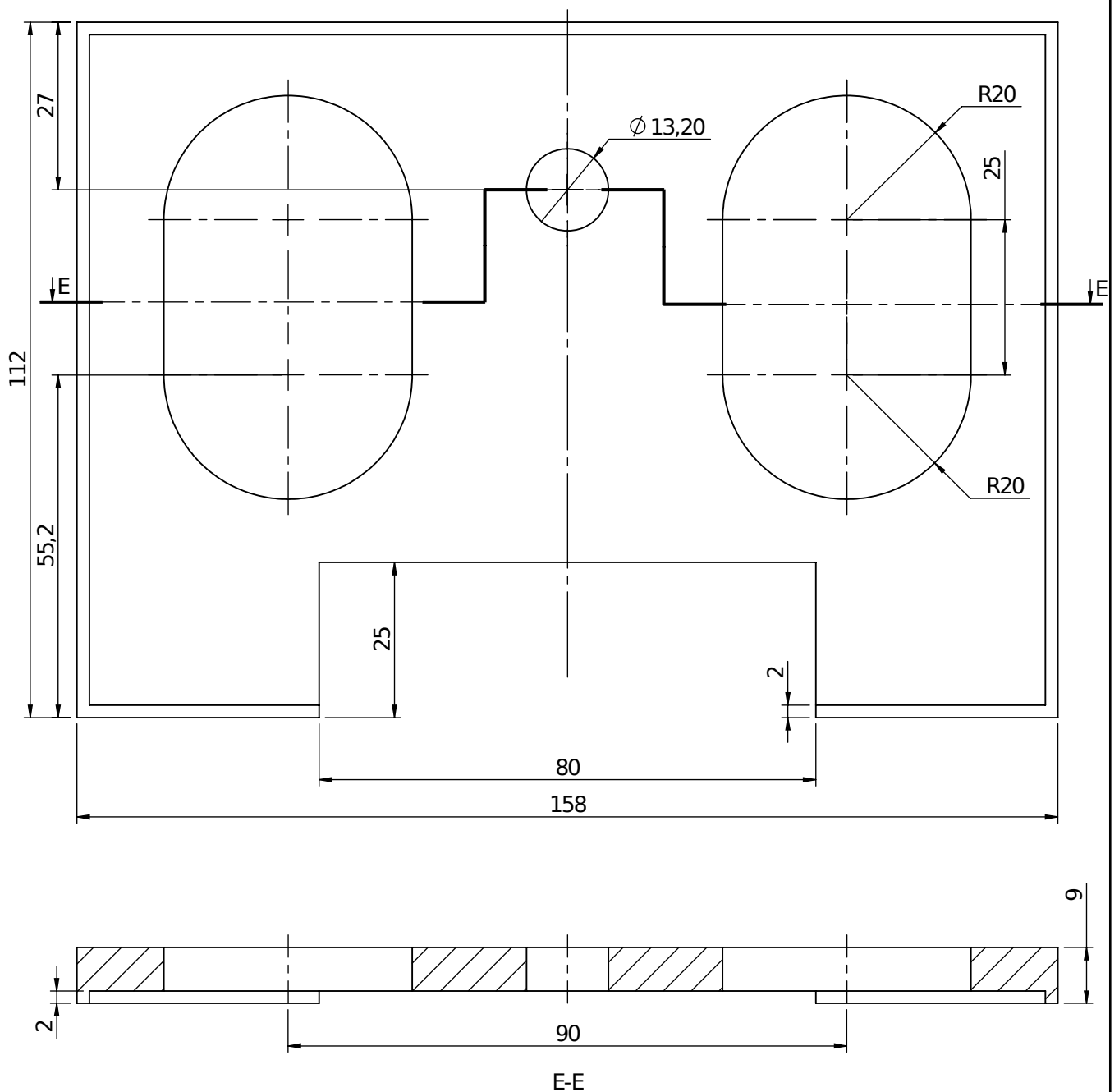
5 / 7

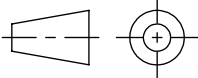
Fecha:

Agosto de 2016



	<p>TFM:</p> <p>Diseño e implementación de un sistema de control para la representación gráfica a partir de imágenes</p>	<p>Autor:</p> <p>Lois Rilo Antelo</p>	<p>Material</p> <p>Plástico ABS</p>	<p>Escala</p> <p>1:1</p>
<p>Curso:</p> <p>2015/16</p>		<p>Denominación:</p> <p>Pinza</p>		<p>Plano:</p> <p>6 / 7</p> <p>Fecha:</p> <p>Agosto de 2016</p>



 <p>Curso: 2015/16</p>	<p>TFM:</p> <p>Diseño e implementación de un sistema de control para la representación gráfica a partir de imágenes</p>	<p>Autor:</p> <p>Lois Rilo Antelo</p> <p>Denominación:</p> <p>Soporte para controladoras</p>	<p>Material</p> <p>Plástico ABS</p> <p>Plano:</p> <p>7 / 7</p> <p>Fecha:</p> <p>Agosto de 2016</p>	<p>Escala</p> <p>1:1</p>
---	---	--	--	--------------------------